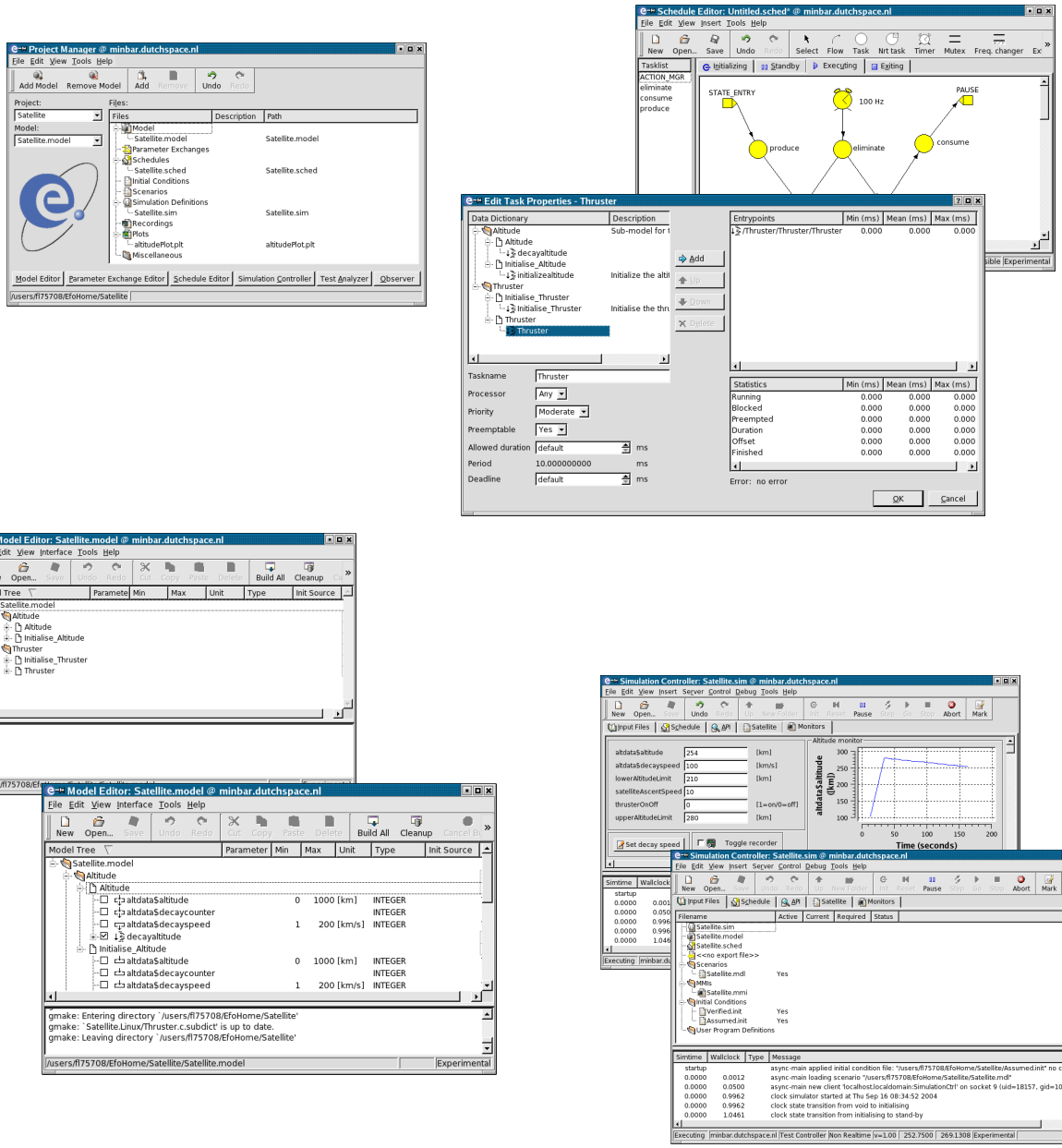
EuroSim Mk4.3
EuroSim Manual Pages



National Aerospace Laboratory NLR

Dutch Space
an EADS Astrium company

NSPYRE
making technology matter

Summary

EuroSim Mk4.3 is an engineering simulator to support the design, development and verification of space (sub) systems defined by ESA programmes of various scales. The facility provides a reconfigurable real-time execution environment with the possibility of man-in-the-loop and/or hardware-in-the-loop additions.
This document contains the printed manual pages for EuroSim end-users. It can be used as an annex to the Software User Manual.

Revision Record

Issue	Revision	Date	Reason for change	Changes
2	1	28-Feb-1997	Document creation, contains chapters 4.1-4.4 of the System ICD FSS-EFO-SPE-134 2/0.	All pages.
3	0	2-Sep-2004	This document is now automatically built from the EuroSim manual pages.	All pages.
4	0	18-Apr-2006	Regenerated from the EuroSim manual pages.	All pages.
4	1	28-Jan-2008	Regenerated from the EuroSim manual pages.	All pages.

Contents

Chapter 1

Introduction

1.1 Purpose

The purpose of this document is to provide a printed copy of the EuroSim manual pages to the end-user. It can be used as a reference to the EuroSim Software User Manual.
Releases of EuroSim always contain an electronic read-only copy of the latest revision of the manual pages.

1.2 Scope

This document contains only the printed manual pages relevant to EuroSim end-users. It contains manual pages for all EuroSim tools that can be started from the Unix command line, and libraries that can be used from within EuroSim model code.

Chapter 2

Manual pages

The manual pages in the following sections are divided firstly according to their type, following the System V man page division:

- man 1: user commands;
- man 3: library functions;
- man 4: special files and file formats;
- man 8: daemons, documentation and maintenance commands.

2.1 man1: user Commands

This section of manual pages contains all ‘end user commands’ common to each EuroSim distribution. These commands are typically found in the $EFOROOT/bin directory.

2.1.1 a2r

Convert a EuroSim ASCII recorder file into a binary recorder file

SYNOPSIS

a2r [**-o** *outputfile*] [**-d** *dictname*] [**-D**] [**-t** *timefmt*] [*ascii-recordfile*]
a2r [**-h**]

DESCRIPTION

a2r converts an *ascii-recordfile* into a binary formatted one. This is a simple one to one translation.
If no *ascii-recordfile* is given **a2r** reads from standard input. If **r2a** is used to produce the *ascii-recordfile* the normal header format should be used (which is the **r2a** default).

OPTIONS

-o *outputfile*

The given file should be used for output (instead of the default standard output).

-d *dictname*

The specified dict should be used instead of the dict listed in the inputfile header.

-t *timefmt*

Simulation time input format, which is similar to **strftime**(3C) extended with %[.|0][precision]s for the subseconds, such that %.3s reads milliseconds and %6s reads a microsecond value. If the format = %f floats will be read. (default: %Y-%m-%d %H:%M:%S.%s if simulation time is UTC, else float.)

-D

Do not use the dict found in the header of the file for converting the data.

-h

Print usage message and quit.

SEE ALSO

r2a(1), **dictrec**(4)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl

AUTHOR

Robert H. de Vries, Kees van der Poel, Dutch Space BV, The Netherlands.

2.1.2 api2dict

Generate Dict from API header in C, Ada or Fortran 77 source file

SYNOPSIS

api2dict [**-l** *suffix*] [**-u** *id*] [**-o** *dict*] [**-D***flags*] *source-file*

DESCRIPTION

api2dict parses API compliant model code to extract the information that is necessary to generate a *dict*. The API information is stored in commented headers. **api2dict** extracts the information from these headers, containing publicly marked data variables and entry points. The format of the API header is described in **api**(4).

OPTIONS

-o *dict*

The generated dict is written to the file *dict*. If this option is omitted, *dict* is written to the standard output.

-u *id*

Dummy compatibility flag with **code2dict**(1) and **subm2dict**(1).

-l *suffix*

Optional *suffix* flag for specifiying language type if code is written from standard input instead of read from file. Use the *c*, *adb*, *f* suffices you'd normally use.

-D*flags*

Debug. Specify debug flags depending on the level of detail you want. Only available if **api2dict** was compiled with the **-DDEBUG** option.

DIAGNOSTICS

The following error messages can be given:

Duplicate source-file.

Only one *source-file* should specified.

Duplicate outputname option.

The option **-o** should be given at most once.

File [*source-file*] to be parsed and outputname are identical.

Specifiy another outputname for *dict*

Unknown language suffix '*x*'.

The suffix *x* is not recognized/supported by EuroSim.

Missing language suffix.

Without a derivable suffix **api2dict** can't determine comment convention of the source-file.

syntax of another language detected

In the API header a wrong coding convention is used eg for C-code *int* var(2:3)

Error parsing API header of source-file: ***source-file***

The API parser detected an error. Most likely, this message is preceded by one or more messages from the API parser.

Other diagnostics are self-explanatory, or indicate internal errors.
On error, **code2dict** exits with status not 0, else 0.

ACKNOWLEDGEMENTS

The Windows version of api2dict uses RSA Data Security, Inc. MD5 Message-Digest Algorithm.

SEE ALSO

dict2api(1), **subm2dict**(1), **dict**(3), **api**(4)

NOTES

api2dict does not output a dict with full definitions for user defined types (C or ADA) or common block offsets (Fortran). The reason for this is simple: these definitions are not part of the **api**(4) header, and can *thus* never be the output by **api2dict**(1).
Use **subm2dict**(1) instead.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

Author

Ad Schijven, Dutch Space BV, The Netherlands.

2.1.3 CalibrationEditor

Run the calibration editor

SYNOPSIS

CalibrationEditor [*calibrationfile*]

DESCRIPTION

CalibrationEditor runs the calibration editor to view, create and/or update a calibration file.

SEE ALSO

Read the EuroSim Software User Manual (SUM) for details on how to operate the **CalibrationEditor**.
esimCalibration(3)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert de Vries, Dutch Space BV

2.1.4 code2dict

Generate Dict from Ada, C or Fortran 77

SYNOPSIS

code2dict [**-1**] [**-l** *suffix*] [**-u** *id*] [**-o** *dict*] [**-b**] [[**–**] *parser-argument...*]

DESCRIPTION

The *code2dict*(1) command invokes the *pars2dict*(1) command, with possible pre-processing and remapping of options. The following language suffices are recognized: .adb .ads .ada (Ada), .i .c (C), .f .F (Fortran 77). For suffices **.c** and **.F**, the source is first piped through the C pre-processor using "cc -E"; for C and Fortran, this can be overridden by an explicit **-cpp** or **-nocpp** option among the parser arguments.

OPTIONS

All regular options behave the same as their counterparts in *pars2dict*(1), but the parser argument options are treated differently.
For Ada, only options starting with **-D** are supported, since it calls *api2dict*(1) rather than a true parser.
For Fortran, extensions are switched on, and tables set, with the implicit extra options: **-Xextensions=1 -Xcontinuations=99 -Nq1000 -Nn5000**. The following option patterns are quietly ignored: **-32**, **-n32**, **-D?***, **-KPIC**, **-O***, **-U?***, **-cckr**, **-cord**, **-diag_*** *arg*, **-feedback** *arg*, **-framepointer** *arg*, **-g**, **-g[0123]**, **-kpicopt**, **-mips[1234]**, **-nokpiopt**, **-non_shared**.
The following option patterns are passed through unchanged, or mapped as indicated in parentheses: **-66** (-6), **-C**, **-I***, **-N***, **-U** (-U), **-cpp** (see above), **-d_lines** (-Xd_lines=1), **-i2**, **-i4**, **-nocpp** (see above), **-onetrip** (-Xonetrip=1), **-u**, **-w**, **-w66**.
The options **-64 -i8 -r8** are not supported (yet).
For (ISO-)C, extensions are switched on, with the implicit extra options: **-Xwarnings=0 -Xstricts=0**. The following option patterns are quietly ignored: **-D?***, **-I***, **-KPIC**, **-O***, **-U?***, **-Xcpluscomm**, **-acpp**, **-ansi**, **-ansiposix**, **-cckr**, **-cord**, **-diag_*** *arg*, **-feedback** *arg*, **-framepointer**, **-g**, **-g[0123]**, **-kpicopt**, **-mips[1234]**, **-nokpiopt**, **-non_shared**, **-nostdinc**.
The following option patterns are passed through unchanged, or mapped as indicated in parentheses: **-common** (-m), **-cpp** (see above), **-nocpp** (see above), **-w**.
The options **-32**, **-64** and **-n32** make c2pars assume the appropriate datamodel in accordance to the corresponding options of the native MIPS C compiler.

SEE ALSO

pars2dict(1).

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

2.1.5 dict2api

Generate an API header from a Dict

SYNOPSIS

dict2api [**-o** *source-file*] [**-l** *language*] [**-e**] [**-s**] [**-V**] [**-1**] *dict*

DESCRIPTION

This command is part of the ADD object ApiEditor.
Dict2api reads the file *dict* or the standard input if no *dict* argument is given, to obtain a Dict. The hierarchy in this Dict is searched to determine the source language of the program from which this Dict was generated. From a file reference node in the hierarchy, the source language is determined as follows. If the file name ends with an *a* it is assumed to be written in Ada; if the file name ends with a *c* or an *i* it is assumed to be written in C; if it ends with an *f* it is assumed to be written in Fortran 77. From a variable node the source language can be determined by looking at the variable description table.
Dict2api then extracts information about variables and entries from the Dict and writes it in the form of an API header using the comment convention of the programming language. Note that variables which are **Global_State_Variables** will not appear as **Global_Input_Variables** or **Global_Output_Variables**, although they may be read and/or written by a function that appears as an **Entry_point**.
Some variable names in the API header will differ from the corresponding variable names in the Dict. If a variable name starts with the prefix __**u**, this prefix and any following digits will not become part of the variable name in the API header. If the source language is Fortran 77, a variable name containing a $ is assumed to be a COMMON variable name. The part of the name before the $ is the COMMON block name, and the part after the $ is the variable name proper. The variable name is printed in the API header as the COMMON block name within slashes (/), followed by the variable name proper. If the COMMON block name is _BLNK_ it is omitted, i.e. the part within the slashes is empty.

OPTIONS

-o *source-file*

The generated API header is written to the file *source-file*. If this option is omitted it is written to the standard output.

-l *language*

This option overrides the automatic language selection. Use c for ANSI-C, use f for Fortran 77 and a for Ada.

-e

Only if this option is given, the generated API header for Fortran 77 and Ada ends with an empty line.

-s

Set this flag when generating API for Mk3. Mk2 had only the API dict as input. In Mk3 the tool uses the subdict with flags indicating that a variable or entrypoint is in the API.

-V

Print version identification on error output and exit.

-1

Print API header in (old) EuroSim Mk01 form. This option is off by default.

DIAGNOSTICS

The following warnings can be given:

Warning: *variable* has no filescope

A variable name starting with __**u** is found, but the variable is not a C static variable.

Warning: *variable* has no prefix

A C static variable is found, but the name doesn't begin with __**u**.

The following error messages can be given:

Duplicate outputname

The **-o** option is used more than once.

Usage: dict2api [-l langtype] [-e] [-o source-file] [-s] [-1] [-V] [dict]

An unknown option, or more than one *dict* argument, have been given.

Source language can't be determined

There are no file reference and variable nodes, or the specified source language is not the same for all file references and variables.

Other diagnostics are self-explanatory, or indicate internal errors.
On error, **dict2api** exits with status 1, else 0.

SEE ALSO

dict(3), **pars2dict**(1), **dict2pub**(1), **api**(4)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

2.1.6 dict2pub

Generate C, Fortran, or Ada publish code from a Dict

SYNOPSIS

dict2pub [**-i** *prepend-source* | **-I**] [**-l** *langtype*] [**-u** *id* | **-U**] [**-p** *prefix*] [**-o** *output-source-file*] [**-e**] [**-s**] [**-V**] [*dict*]

DESCRIPTION

This command is part of the ADD object PreCompilers.
Dict2pub reads the file *dict* or the standard input if no *dict* argument is given, to obtain a Dict. The hierarchy in this Dict is searched to determine the source language of the program from which this Dict was generated. From a file reference node in the hierarchy, the source language is determined as follows. If the file name ends with an *a* it is assumed to be written in Ada; if the file name ends with a *c* or an *i* it is assumed to be written in C; if it ends with an *f* it is assumed to be written in Fortran 77. From a variable node the source language can be determined by looking at the variable description table.
Dict2pub then writes a procedure definition in the appropiate programming language, containing code to update the data dictionary at runtime. For each variable in the Dict, the necessary declarations and statements are written, to put the runtime address of the variable in the data dictionary.
For C and Ada, no declarations are needed to make the variables in the Dict accessible by the procedure that is created. For Fortran, the prodecure starts with declarations for all COMMON blocks that were used to define the variables in this Dict. The runtime address is updated with a call to the dict library routine **dictgpublish**_(3).

OPTIONS

-i *prepend-source*

The generated file will have the file *prepend-source* prepended.

-I

The generated file will have the source file, from which this *dict* has originally been made, appended.

-l *langtype*

Specifies the source language if this can't be obtained from the input dict. *langtype* Can be one of **f**, **c**, or **a**, for Fortran, C, or Ada, resp.

-u *id*

The generated procedure will be named **esimpub**_*id*; (see also option **-p** below).

-U

The unique *id* described above will be fetched from the dict. If neither **-u**, nor the **-U** option has been given, the unique *id* described above will be empty.

-p *prefix*

Replace the default prefix for the generated procedure (**esimpub**_) by the given *prefix*.

-o *output-source-file*

The generated publish code is written to the file *output-source-file*. If this option is omitted it is written to the standard output.

-e

Also generate code for publishing entrypoints.

-s

Only generate code for data and entrypoints in the API header.

-V

Print version identification on error output and exit.

DIAGNOSTICS

The following warnings can be given:

Warning: *variable* has no filescope

A variable name starting with __**u** is found, but the variable is not a C static variable.

Warning: *variable* has no prefix

A C static variable is found, but the name doesn't begin with __**u**.

The following error messages can be given:

Duplicate outputname

The **-o** option is used more than once.

Duplicate id

The **-u** or **-p** option is used more than once.

Usage: dict2pub [-I|-i prepend-source] [-l langtype] [-U|-u id] [-e] [-s] [-p prefix] [-o output-source-file] [-V] [dict]

An unknown option, or more than one *dict* argument, have been given.

Source language can't be determined

There are no file reference and variable nodes, or the specified source language is not the same for all file references and variables.

Other diagnostics are self-explanatory, or indicate internal errors.
On error, **dict2pub** exits with status 1 and tries to remove the generated output file, else **dict2pub** exits with status 0.

SEE ALSO

dict(3), **pars2dict**(1), **dict2api**(1)

BUGS

The generated Ada is gnat(1) specific.
The generated Fortran code uses non-standard extensions: underscores in identifiers, and identifiers longer than 6 characters. Mail bug reports to esim-bugs@dutchspace.nl.

2.1.7 dictcat

Display the contents of dictionaries

SYNOPSIS

dictcat [**-a**] [**-f** *filter*] [**-o** *out-file*] [–] [*dict...*]
dictcat -?

DESCRIPTION

dictcat Displays hierarchically the contents of one or more dictionaries, see also ***dict*(3)**.
dict... denotes the dictionaries (i.e. paths) to display. The special value - means that the dictionary will be read from the standard input channel. This will also be done when no *dict* argument at all has been specified.

OPTIONS

-f *filter*

This filter specifies the type(s) of nodes in the hierarchy of the dictionary to display. It is a comma-separated list of **DTreeType** values, with the common *dIs* prefix removed (see also **dicthier**(3)). To view a list of possible values, ask the **dictcat** program for a *usage* message (see the **-?** option below).

-o *out-file*

The output will appear on file *out-file*, instead of standard output.

-a

This flag will cause structures to be expanded, rather then plain variables to be written.

-?

Asks explicitely for a *usage* message.

DIAGNOSTICS

When an error occurs, this tool will try to produce a diagnostic as applicable as possible.
On error, **dictcat** exits with status 2, else 0.

EXAMPLE

Print hierarchically all nodes that are of type **dIsFileRef** or **dIsVar**, with the output redirected to file **foo**, from the dictionary in file **bar.dict**:

```
example% dictcat -f FileRef,Var -o foo -- bar.dict
```

SEE ALSO

dict(3), **dicthier**(3)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

2.1.8 dictdict

Convert (old) EuroSim datadictionary to datadictionary

SYNOPSIS

dictdict [*options*] [–] *dict-file*

DESCRIPTION

dictdict converts a possibly old EuroSim datadictionary specified by *dict-file* to the latest version under control of the following *options*:

-b *suff*

specifies backup suffix (i.e. extension), instead of '.orig'

-o *outfile*

specifies output file, instead of *dict-file*.

-f

force; overwrite backup file

-r

report default sorting mode while converting

-s *sortmode*

set sorting mode

SEE ALSO

dict(3).

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Jan Hein Buhrman & Wim Elhorst (Dutch Space BV)

2.1.9 dictdumps

Convert initial condition files and snapshots

SYNOPSIS

dictdumps [*options*] [–] *dump-file [dump-file ...]*

DESCRIPTION

dictdumps converts a (set of) initial condition or snapshot *dump-file*s to **one** snapshot or initial condition file under control of the following *options*:

-o *outfile*

specifies output file, instead of standard output *stdout*.

-d *dict-file*

specifies data dictionary to use, rather than the one specified in the header of the *first dump-file* specified.

-v

report progress to standard error *stderr*.

-b

write binary (snapshot) output rather then ascii (initial condition) output.

-f

write the full snapshot/init condition file of the datadictionary. This means all variable nodes in the datadictionary hierarchy. Default behaviour is to write the values of variables that were explicitly *set* in one the *dump-file*s because they matched a variable in the data dictionary used.

-c

write only the variables whose values changed with respect to the datadictionary default values.

If in the set of *dump-file*s the same variable is set to different values, the last value will be in the output.

If variables occur in one of the *dump-file*s which are unavailable in the referenced data dictionary, then **dictdumps** complains about these variables.

SEE ALSO

dict(3).

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Wim Elhorst (Dutch Space BV)

2.1.10 dictmerge

Merges dicts

SYNOPSIS

dictmerge [**-m** *modelfile*] [**-r** *reference*] [**-o** *dict*] *dict* ..

DESCRIPTION

This command is part of the ADD object HeaderParsers.
dictmerge merges one or more *dicts* generated from compilation units.
dictmerge verifies if the *dict* is **consistent/complete** : all entries and data items, and its type, in the hierarchy are present.
It verifies if the *dict* to be merged doesn't **conflict** with the *dicts* merged sofar. Are entries and data items with local scope unique, do 'global' data items have the same attributes and type specification, etc.

OPTIONS

-o *dict*

The generated Dict is written to the file *dict*. If this option is omitted the Dict is written to the standard output.

-m *modelfile*

modelfile is used to obtain modelpaths (hierarchy path) from which sub-dicts rooted. The contents of the dicts is attached to the found hierarchy paths. (The source file freetext node (*__SUBM_filepath*) is used as a key to find the hierarchy path. (See also **dict**(3)).

DIAGNOSTICS

The following error messages can be given:

Duplicate outputname option.

The option **-o** should be given at most once.

File [*dict*] to be merged and outputname are identical.

Specifiy another outputname for *dict*

Inconsistent dict read *dict*

An inconsistency in the *dict* is detected. Most likely, this message is preceded by one or more comprehensive messages.

Merge conflicts occured *dict*

Merge conflicts were detected, when trying to add *dict* to the sofar generated Dict. Most likely, this message is preceded by one or more comprehensive messages.

Other diagnostics are self-explanatory, or indicate internal errors.
On error, **dictmerge** exits with status not 0, else 0.

SEE ALSO

dict(3)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Ad Schijven, Dutch Space BV, The Netherlands.

2.1.11 efoKill

Kill running EuroSim simulators

SYNOPSIS

efoKill [**-***signal*] [**-h** *hostname*] **pid** | **simname**

DESCRIPTION

efoKill kills a running simulator on the current host or the host specified on the command line. A user is only allowed to kill his/her own simulator. Root may kill any simulator. By default SIGTERM is sent to the simulator. The user may specify either the pid as list by *efoList* or the basename of the simulator (without the .exe extension). If there is more than one simulator with the specified name running, the first one is selected as listed by *efoList*

OPTIONS

-signal

Use another signal than the default (SIGTERM). The signal value must be specified numerical.

-h ***hostname***

Select a different host than the current one to kill a simulator on.

NOTES

This program uses RPC-calls to query the EuroSim daemon.
This program does not work with Mk0.1 simulators. They do not set the pgid. The pgid is used to kill all sprocs in one go.

SEE ALSO

efoList(1), **esimd**(8)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

R.H. de Vries

2.1.12 efoList

List running EuroSim simulators

SYNOPSIS

efoList [**-a**] [**-l**] [**-h** *hostname*]

DESCRIPTION

efoList lists running simulators on the current machine, or the specified machine or on all machines connected to the local network. It can list a summary or a full dump of all information known about the simulator.

OPTIONS

-a

List simulators running on all machines on the local network.

-l

Make a long listing

-h *hostname*

List simulators running on host *hostname* or on the currents host if this option is not specified

NOTES

This program uses RPC-calls to query the EuroSim daemon.

SEE ALSO

efoKill(1), **esimd**(8)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

R.H. de Vries

2.1.13 esim

Run the EuroSim project manager

SYNOPSIS

esim

DESCRIPTION

esim runs the project manager. Use the project manager to logically group files that are needed for a simulation run (model file, schedule file, simulation definition file, etc.).

FILES

projects.db

The projects database file.

ENVIRONMENT VARIABLES

EFO_HOME

Specifies the location of the projects database file (**projects.db**). If the **EFO_HOME** environment variable is not set, the projects database file is accessed (and created) in the .eurosim sub-directory of the users home directory.

SEE ALSO

Read the EuroSim Software User Manual (SUM) for details on how to operate the project manager.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Fred van Lieshout, Dutch Space BV

2.1.14 esim2html, esim2pdf, esim2ps

Programs to convert EuroSim data to a HTML, portable document format (pdf) or postscript (ps) file.

SYNOPSIS

esim2html | **esim2pdf** | **esim2ps** [**-h**] [**-o** *output file*] [**-m** *model file*] [**-d** *data dictionary file*] [**-s** *schedule file*] [**-i** *initial condition file*] [**-a** *MDL file*] [-**M**] [-**P**]

DESCRIPTION

This program reads files with EuroSim information (given by their **filename(s)** and converts the contents of these files to a HTML document. The files can be either:

The data dictionary and MDL actions files (<name>.mdl).

The other filenames will be retrieved from these files.

A subset of the following individual files:

Model file
Data dictionary file
Schedule file
Initial conditions file
MDL actions file

(**warning**: The data dictionary file must always be in the subset)

The HTML document consists of at maximum 5 subdocuments (one for each file) and contains internal links between the subdocuments.

OPTIONS

-h

Display a help message

-o *output file*

Send result to **output file**. If this flag is not used the output goes to the standard output.

-m *model file*

Model file whose contents are put in the generated document.

-d *data dictionary file*

Data dictionary file whose contents are put in the generated document.

-s *schedule file*

Schedule file whose contents are put in the generated document.

-i *initial conditions file*

Initial conditions file whose contents are put in the generated document.

-a *MDL file*

MDL actions file whose contents are put in the generated document.

-M

Do not process model description file(s) referenced in *model file*.

-P

Do not process parameter exchange file(s) referenced in *schedule file*.

EXAMPLE

This example reads data from a model and a data dictionary file and puts the results in a HTML document named test.html:

```
esim2html -d test.dict -m test.model -o test.html
```

SEE ALSO

html2ps(1), **ps2pdf**(1)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Bart Schipperijn (Dutch Space BV).

2.1.15 esimcapability

EuroSim capability database frontend

SYNOPSIS

esimcapability [*options*...] [*capabilities*...]

DESCRIPTION

esimcapability prints certain information about the EuroSim capability database. Without *options* and *capabilities* all available database keys in the default capability file are printed.
Otherwise, the information that is displayed is controlled by the selection of *options*.
The capability keys can be used with *options* to extract information for location of libraries, include directories, define directives and even code generation.
Options that can always be used:

-f *name*

Read the given capability file *name* and get all requested info from it.

options is either one of:

-d

print description of given capabilities

-c

print code for using the given capabilities

-or- *options* is a combination of:

-I

print include directories for given capabilities

-D

print define directives for given capabilities

-L

print library directories for given capabilities

-l

print libraries for given capabilities

-v

print values for given capabilitie(s)

-V

print key=value pairs for given capabilitie(s)

-with optionally- the following modifier *options*:

-t

print default capability

-m

print ala make (brackets around envvars)

-e

EXPAND all envvars

-E

but keep EFOROOT

FILES

$EFOROOT/etc/EuroSim.capabilities contains the 'default' capabilities.

SEE ALSO

auxcapability(3), **esimcapability**(4).

BUGS

esimcapability does not return an appropriate error code when information is requested about an non-existing cability/key.
Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Wim Elhorst (Dutch Space BV)

2.1.16 esimversion

Return EuroSim version

SYNOPSIS

esimversion

DESCRIPTION

esimversion prints the version of the EuroSim installation first found in your path. That's all.

Bugs

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Wim Elhorst (Dutch Space BV)

2.1.17 esimsh

EuroSim shell

SYNOPSIS

esimsh

DESCRIPTION

esimsh is an interactive shell to control EuroSim. It is based on perl. It uses an advance command line editor based on the readline(3) library which is also used in bash(1).
The goal of the tool is to make it easier to create batch scripts. The tool offers a command to log every command to a file. The resulting file can be used as a starting point for a non-interactive batch script.
There is extensive command line completion support. Every command can be completed, as well as data dictionary variables, MDL script names, action names and file names.
The goal of the tool is to make it possible to automate every possible simulation run. It is therefore possible to generate new MDL files (EuroSim::MDL(3)) and initial condition files (EuroSim::InitCond(3)).
Starting and manipulating a simulation run is done with the functions available in the EuroSim::Session(3) module. The EuroSim::Conn(3) module offers mostly low-level message sending functions which are used by the EuroSim::Session(3) module. The EuroSim::Dict(3) module offers data dictionary manipulation and query functions. The EuroSim::Link(3) module offers the TmTc link functions.

BUGS

Not all commands offer command line completion for all arguments.
Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert H. de Vries, Dutch Space BV

SEE ALSO

perl(1), EuroSim::Session(3), EuroSim::Conn(3), EuroSim::Dict(3), EuroSim::Link(3), EuroSim::InitCond(3), EuroSim::MDL(3), Term::ReadLine::Gnu(3), readline(3).

2.1.18 gendatapool

Generate the datapool source code and data dictionary file

SYNOPSIS

gendatapool [**-e** *errinjfile*] [**-d** *dictfile*] [**-n** *rootname*] [**-o** *output basename*] [**-v**] *MODELDESCRIPTION-FILE(S)*

DESCRIPTION

gendatapool generates the source code file which contains the datapool variables and entrypoints and runtime publication code. At the same time it generates the (sub) data dictionary file for the variables and the entry points.
The tool is invoked by the EuroSim simulator executable build system. It is normally not invoked directly by an end-user.

OPTIONS

-d *errinjfile*

Specify the error injection description file.

-d *dictfile*

Specify which data dictionary file to use.

-n *rootname*

Specify the name of the root org node for the datapool (default is 'datapool').

-o *output basename*

Specify the output base file name (default is 'datapool'). The generated files are called *datapool*.c and *datapool*.subdict.

-v

Run **gendatapool** in verbose mode.

SEE ALSO

ModelDescriptionEditor(1)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Fred van Lieshout, Dutch Space BV

2.1.19 genversion

Generate version trace for a model file

SYNOPSIS

genversion [**-c**] [**-r** *version*] *modelfile* [*repository versionsystem*]

DESCRIPTION

This command reads (a version of) a model file and produces a version trace on standard output. The -c flag generates an ANSI-C string variable definition ready for inclusion into an executable. Without this flag, the lines need to be included (possibly as comment into the derivative files (executable, *dict* or *schedule* files). Each line is formed recognisable by both what(1) and ident(1):

```
@(#)$Trace: ... $
```

With regard to versioning and tracing, a model can be in three states: *reproducible*, *traceable* and *not_traceable*.
A model is in a *reproducible* state if no requirements are left blank and if the contents of each of the constituent files corresponds to their required version. A *reproducible* model is reproducible using the model file name and version only.
A model is *traceable* if the model file and all of its constituent files are versioned and unmodified, but some requirements are left blank or if the contents of some of the files deviate from their requirement. A *traceable* model is reproducible using the model file name and version and correcting deviant files using the additional trace lines.
A model is *not_traceable* if the model file or any of its constituent files is unversioned or modified. A *not_traceable* model is non-reproducible because the contents of some of the files cannot be identified or reproduced after modification.
In addition to a trace line for the model file, trace lines are output for each constituent file that has no requirement set:

```
@(#)$Trace: sensors.c (nonexisting) $
@(#)$Trace: sensors.c (unversioned) $
@(#)$Trace: sensors.c (2.3) $
@(#)$Trace: sensors.c (2.3+) $
```

and for each constituent file that deviates from its requirement:

```
@(#)$Trace: sensors.c 2.3 (nonexisting) $
@(#)$Trace: sensors.c 2.3 (unversioned) $
@(#)$Trace: sensors.c 2.3 (1.1) $
@(#)$Trace: sensors.c 2.3 (2.3+) $
```

If a version of the model file is specified on the command line (optional) then the references to source files in that version (rather than the default of the local model file) are compared with the current working versions.
If the repository and versionsystem is not specified, genversion will try to detect them for you.

OPTIONS

-c

Generate ANSI-C code instead of plain text

-r *version*

Use this version of the modelfile instead

modelfile

Model to query

repository

Location of the repository

versionsystem

Specify which version system is in use by the repository. Supported are cvs and cadese.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert de Vries, Dutch Space BV

2.1.20 mkcall_sched

Make call_sched() Ada function

SYNOPSIS

mkcall_sched [*submodel ...*]

DESCRIPTION

This command is part of the ADD object PreCompilers. **Mkcall_sched** generates a function **mkcall_sched** to invoke the scheduler's main loop. The function is preceded by a list of *"with"* clauses for all the submodels' modules as given in the submodel list; this will force initialization of those modulesmkcall_sched.1.pod to take place before calling the **mkcall_sched** function itself. Hence, calling the function will imply module initialization followed by entering the scheduler.
Mkcall_sched writes the generated Ada code to standard output. There are no possible diagnostics.

Caveats

The generated code contains pragmas that make it rather compiler-dependent. Currently, ***gnat*(1)** is assumed.

SEE ALSO

dict2pub(1).

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

2.1.21 mkpub

Generate C function that calls publish routines in precompiled files

SYNOPSIS

mkpub [*file* ...]

DESCRIPTION

This command is part of the ADD object PreCompilers.
If a file name ends with an *adb* it is assumed to be written in Ada; if the file name ends with a *c* or an *i* it is assumed to be written in C; if it ends with an *f* of *F* it is assumed to be written in Fortran 77.
mkpub then writes on standard output a C function definition, containing calls to the publish routines defined in the files. The source language of each file is used to determine what the definition of the publish routine looks like. The name of the generated C function is **all_esimpub**.

DIAGNOSTICS

The following warnings can be given:

Error: *file*: no publication function found

mkpub couldn't find a definition of a publish routine.

The following error messages can be given:

Error: *file* has unknown suffix

The suffix is not one of *adb*, *c*, *i*, *f* or *F*.

Error: *file*: multiple publication functions found

Only one definition of a publish routine per file is allowed.

Other diagnostics are self-explanatory, or indicate internal errors.
On error, **mkpub** exits with status 1, else 0.

SEE ALSO

dict2pub(1)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

2.1.22 ModelDescriptionEditor

Run the model description editor

SYNOPSIS

ModelDescriptionEditor [**-m** *modelfile*] [**-r**] [*modeldescriptionfile*]

DESCRIPTION

ModelDescriptionEditor runs the model description editor to view, create and/or update a model description file. A model description file specifies which model API variables should be copied to/from the datapool.
The use of the **ModelDescriptionEditor** is optional, but you would typically use model description files when integrating several independent models into one simulator without wanting to do the integration explicitly in (model) source code. Use model description files in combination with parameter exchange files (created by the **ParameterExchangeEditor**) to exchange data between models without or with minimal changes to the model source code. The **ModelDescriptionEditor** can be used to create one or more model description files that describe copies of API variables that exist in the data dictionary. The data dictionary itself is built by the build process (make) that can be started from the **ModelEditor**.

OPTIONS

-m *modelfile*

Specify which model file to use.

-r

Start the **ModelDescriptionEditor** in read-only mode (file cannot be modified).

SEE ALSO

ParameterExchangeEditor(1). **gendatapool**(1). **ModelEditor**(1). Read the EuroSim Software User Manual (SUM) for details on how to operate the **ModelDescriptionEditor**.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Fred van Lieshout, Dutch Space BV

2.1.23 ModelEditor

Run the model editor

SYNOPSIS

ModelEditor [*modelfile*]

DESCRIPTION

ModelEditor runs the model editor to view, create and/or update a model file. A model file specifies which (sub-) model source code files make up the simulator model. The **ModelEditor** can also be used to build the simulator executable and data dictionary. For that purpose it calls **ModelMake** to generate a makefile.

SEE ALSO

ModelMake(1). Read the EuroSim Software User Manual (SUM) for details on how to operate the **ModelEditor**.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Fred van Lieshout, Dutch Space BV

2.1.24 ModelMake

Generate makefile from a model file

SYNOPSIS

ModelMake *model makefile* [**-D**]

DESCRIPTION

ModelMake generates a *makefile* from a *model* file. The generated file contains the list of sources in the model file, the build options (compiler flags, user libraries, include paths, etc...) and compiler specifications as set in the ModelEditor build options dialog box.
The rules to generate the model executable and the other files are included. The file template.make relies heavily on the features of GNU make. It cannot be used with other make programs.
ModelMake is automatically called from the ModelEditor when the build process is started and the *model* file is more recent than the *makefile*.

OPTIONS

-D

Use the default compiler specifications instead of the ones specified by the user.

EXAMPLE

ModelMake Thermo.model Thermo.make

SEE ALSO

make(1), **ModelEditor**(1), *The GNU Make Manual*

FILES

$EFOROOT/lib/templates/template.make

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert de Vries, Fred van Lieshout, Dutch Space BV

2.1.25 ParameterExchangeEditor

Run the parameter exchange editor

SYNOPSIS

ParameterExchangeEditor [**-m** *modelfile*] [**-r**] [*parameterexchangefile*]

DESCRIPTION

ParameterExchangeEditor runs the parameter exchange editor to view, create and/or update a parameter exchange file. A parameter exchange file specifies which output variable(s) in the datapool should be copied to which input variable(s) in the datapool. Datapool variables are specified by one ore more model description files, which are created by the **ModelDescriptionEditor**.
The use of the **ParameterExchangeEditor** is optional, but you would typically use parameter exchange files when integrating several independent models into one simulator without wanting to do the integration explicitly in (model) source code.

OPTIONS

-m *modelfile*

Specify which model file to use.

-r

Start the **ParameterExchangeEditor** in read-only mode (file cannot be modified).

SEE ALSO

ModelDescriptionEditor(1). Read the EuroSim Software User Manual (SUM) for details on how to operate the **ParameterExchangeEditor**.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Fred van Lieshout, Dutch Space BV

2.1.26 pars2dict

Generate Dict from Ada, C or Fortran 77 source file

SYNOPSIS

pars2dict [**-l** *suffix*] [**-u** *id*] [**-o** *dict*] [**-p** *string*] [**-b**] [**-V**] [**-1**] [**-a**] [[**–**] *parser-arguments*]

DESCRIPTION

This command is part of the ADD object LanguageParsers.
Depending on the suffix given by the **-l** *suffix* option, **pars2dict** calls an Ada, C, or Fortran 77 parser with **popen**(3) to extract the information that is necessary to generate a Dict. If this option is missing, *parser-arguments* must be present, and the last argument is assumed to be a source file. In that case **pars2dict** examines the suffix of the source file to determine its source language, and hence, which parser to call. The suffix, whether given by the **-l** *suffix* option or derived from the source file, determines the source language as follows. If the suffix is **a** or **ada**, **pars2dict** calls the Ada parser, if it is **c** or **i**, **pars2dict** calls the C parser, and a suffix **f** or **ftn** tells **pars2dict** to call the Fortran 77 parser.
If options must be given to the parser, they must be given after **–**, because otherwise they are interpreted as, possibly illegal, options to **pars2dict** itself.

OPTIONS

-l *suffix*

Call the parser, appropriate to *suffix*, as described above.

-u *id*

If this option is given, variables which can be accessed in *source-file* only, e.g. C static variables, are prefixed in the generated Dict with __**u***id*.

-o *dict*

The generated Dict is written to the file *dict*. If this option is omitted the Dict is written to the standard output.

-1

When not specified pars2dict removes variable-paths from the top hierarchy level when that variable is (also) referenced under an entrypoint.

-p *dict*

String to recognize in parser output as generic pointer type. Defaults to ”$pointer”.

-b

Delete variables containing bitfields: emit a warning, and not to put them in the dict. Bitfield are members of type ”$notype”. The default is to leave them in.

-a

Enter procedures with parameters into the dict as entry points, contrary to the normal rule that only parameterless procedures are entry points. This is for backward compatibility.

-V

Print version identification on error output.

ENVIRONMENT

If the environment variable A2PARS is set and non-null, it is assumed to hold the path name of the Ada parser. If the environment variable C2PARS is set and non-null, it is assumed to hold the path name of the C parser. If the environment variable F2PARS is set and non-null, it is assumed to hold the path name of the Fortran 77 parser. Otherwise the Ada parser is assumed to be located in *eforoot***/lib/a2pars**, the C parser is assumed to be located in *eforoot***/lib/c2pars**, and the Fortran 77 parser in *eforoot***/lib/f2pars**, where *eforoot* is the value of the environment variable EFOROOT. If the environment variable EFOROOT is not set, the program fails.

DIAGNOSTICS

The following warnings can be given:

Warning: *variable* in *procedure* will not be entered in the data dictionary

A Fortran 77 variable declared as SAVE or a C variable with block scope declared as static is encountered.

Warning: COMMON variable *variable* has different offsets

The sequence of variables in a Fortran 77 COMMON block is not identical in all subroutines, and/or their types are different. As a result, different procedures use a different memory location for *variable*.

The following error messages can be given:

Usage: pars2dict [*-l suffix*] [*-u id*] [*-o dict*] [*-V*] [*-b*] [[–] *parser-arguments*]

An unknown option has been given, or the required argument was missing.

Duplicate language option

The option **-l** should be given at most once.

Duplicate outputname

The option **-o** should be given at most once.

Duplicate id

The option **-u** should be given at most once.

Language option or source file required

Without one of these, **pars2dict** can't determine which parser to call.

Unknown language suffix '*sfx*'

The suffix *sfx* is not one of **a**, **ada**, **c**, **i**, **f**, or **ftn**.

Parser failed

The parser could not be found, or it could not find a valid program in its arguments or standard input. Most likely, this message is preceded by one or more messages from the parser.

Other diagnostics indicate internal errors, or are messages from the parser.
On error, **pars2dict** exits with status 1, else 0.

SEE ALSO

dict2api(1), **dict2pub**(1), **dict**(3), **popen**(3)

BUGS

The Ada parser hasn't been tested yet.
Mail bug reports to esim-bugs@dutchspace.nl.

2.1.27 pdf2plt

Convert plot definition files in the old TestAnalyzer format (.pdf) to the new plot file format (.plt).

SYNOPSIS

pdf2plt *pdffile*

DESCRIPTION

The pdf2plt command reads a file with a plot definition in the old TestAnalyzer format (a file with extension pdf), converts the contents to the new plot file format and writes them to a file with extension plt.
The following arguments must be given:

pdffile

the name of the file with the old plot definition to be converted.

EXAMPLES

This example reads the (old style) plot definition from the file test.pdf, converts it to a new style plot definition and writes it in the file test.plt.

```
pdf2plt test.pdf
```

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Bart Schipperijn, Dutch Space BV, The Netherlands.

2.1.28 probe2esh

Convert event-probe scripts to perl scripts

SYNOPSIS

probe2esh *probe-script* > *perl-script*
probe2esh < *probe-script* > *perl-script*

DESCRIPTION

probe2esh is a conversion tool for scripts written for the event-probe tool. event-probe is a debugging tool with very limited capabilities. Perl is much more powerful. A host of perl modules have been created to interface the EuroSim libraries and run-time environment.
probe2esh takes the event-probe script and converts it to an equivalent perl script. The result is a script which can be executed stand-alone. It uses the EuroSim perl modules, as can be seen at the top of the script.

NOTES

There is no need for the *connect* command anymore as the launch command connects automatically to the new simulator. The original connect command is replaced by a comment.
The *join_channel* command is also redundant as the launch command will automatically join all channels.
There is also no need for the *filter* command. All events from the simulator are either automatically handled by default handlers, or they are ignored. The original line with the filter command ends up in the resulting script as a comment.
Always check the script for the comments mentioned above and remove them after you have verified the correct working of the script.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert H. de Vries, Dutch Space BV

SEE ALSO

perl(1), EuroSim::Session(3), EuroSim::Conn(3), EuroSim::Dict(3), EuroSim::Link(3), EuroSim::InitCond(3), EuroSim::MDL(3), esimsh(1).

2.1.29 projconv

Convert EuroSim Mk2 project file to Mk3 project databases

SYNOPSIS

projconv *projects file*
projconv *project directory*

DESCRIPTION

The program has two modes. The mode taking the Mk2 projects file will convert the projects file into a Mk3 projects file and will also convert all the EuroSim projects mentioned in that file. The other mode taking a project directory will only convert the contents of one directory. In both cases the conversion tool will create a database in each project directory containing the filenames of the files it has converted if they can be linked to a specific model file. Some extra manual work may be needed to add the missed files to the database.
The conversion of an individual project directory consists of the following actions:

- build clean for every Mk2 makefile found and remove the makefile
- remove Mk2 temporary model files (extension .tm)
- remove Mk2 schedule file from model file
- convert Mk2 schedule files into Mk3 sched files
- convert Mk2 plot description files (pdf) into Mk3 plot files (plt)
- create Mk3 simulation definition files from Mk2 MDL files.
- create a project file database with the following contents: model files, schedule files, simulation defintion files, MDL files, initial condition files, User Program definition files.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert H. de Vries, Dutch Space BV

2.1.30 r2a

Convert a EuroSim binary recording file into an ASCII recording file

SYNOPSIS

r2a [**-o** *outputfile*] [**-d** *dictname*] [**-p** *number*] [**-t** *timeformat*] [**-E**] [**-n**] [**-D**] [**-s**] [**-b**] [**-m**] [**-l** *time*] [**-u** *time*] [**-c** *num-list*] [*recorder file*]
r2a [**-h**]

DESCRIPTION

r2a converts a binary EuroSim *recorder file* into an ASCII format. This is a simple one to one translation. If no *recorder file* is given **r2a** reads from standard input.

OPTIONS

-o *outputfile*

The given file should be used for output instead of stdout

-d *dictname*

The given dict should be used instead of the dict used in the input file

-p *number*

The given number is the precision offset with which the floats are represented. The default is ”0” to make sure no data is lost due to conversion errors.

-t *timeformat*

Simulation time output format this is similar to **strftime**(3C) extended with %[.|0][precision]s for the sub-seconds, such that %.3s prints milliseconds and %6s prints a microsecond value. If the time format = %f floats will be written. (default output: %Y-%m-%d %H:%M%S.%s if simulation time is UTC, else float is written.)

-F <format specifier>

The format specifier overrides the built-in format specification. The format specifier consists of two strings. The first string is the type for which the format is to be overridden. The second string is the printf-style format string. See section **USER DEFINED FORMAT SPECIFICATION** for more details.

-E

The header of the *outputfile* consists of a single line where each column is specified. This can be used for importing data into spreadsheet packages like Microsoft(R) Excel. See section **EXPORT TO MICROSOFT(R) EXCEL** for more details.

-m

Combine multiple (split) recorder files. The assumption is made that the second recorder file name is made up of the basename of the given *recorder file* name appended with ’-001’ followed by the suffix of the given *recorder file* name (if any). If, for example, the first recorder file name that is passed to **r2a** is ’counter.rec’, then **r2a** will search for counter-001.rec, counter-002.rec, and so on up to counter-999.rec. When a file is not found, **r2a** stops processing and does not search any further for other files.

-n

No header is written to the *outputfile*.

-D

Do not use the dict found in the header of the file for converting the data.

-s

don't write extra statistics to output.

-b

write binary data to output.

-l *time*

Specify the lower bound of the simulation time of the data to be converted. The time format is in the form of a floating point number.

-u *time*

Specify the upper bound of the simulation time of the data to be converted. The time format is in the form of a floating point number.

-c *num-list*

Specify the list of columns in the file to be converted. The list is a comma separated list of integer numbers. The column numbering starts at 0.

-h

print usage message and quit.

USER DEFINED FORMAT SPECIFICATION

The following table lists the default format strings for each type:

```
Type name | Default format string
----------+----------------------
int       | %lld
uint      | %llu
fp        | %.*g
lfp       | %.*Lg
```

As you can see there is a precision specifier in the format string for floating point numbers. The precision is specified with the **-p** option. The precision is not used when a user defined format specifier is provided. When you override a default format string, you must provide a format string that is consistent with the parameter that is passed to the printf function. As you can see in the table above the integer types are always of the long long variant (i.e. 64 bits). The floating point number is a double (64 bits) and the long float point number is a long double (96 bits or 128 bits depending on the processor type).

EXPORT TO MICROSOFT(R) EXCEL

To export an EuroSim recorder file to Microsoft(R) Excel 97 SR-1 do the following:

1. On the host where EuroSim is installed run `r2a -E recorder file > r2a.txt`
2. Reboot your system (if double boot system) or go to a system running Microsoft NT or Windows.
3. Start Excel.
4. Select "File:Open...".
5. Select the r2a.txt file by:

a.
go to the proper directory,

b.
select ”Files of Type: All Files”,

c.
select the r2a.txt file,

d.
click ”Open”.

6. Select the original data type by:

a.
select ”Delimited”,

b.
select ”Start import at row: 1”,

c.
select ”File Origin: Windows (ANSI)”,

d.
click ”Next >”.

7. Select the proper delimiter by:

a.
select ”Space”,

b.
click ”Next >”.

8. Select the Column data format

a.
select ”General”,

b.
click ”Finish”.

Excel now reads the file and displays it as a table. Note that Excel has limitations on the number of columns and rows. Excel might either display a rounded-off version of the number or a string of pound signs (#), depending on the display format you’re using.

DIAGNOSTICS

When the program is successful, the program will have converted the given input into correct output. When the program fails, the exit status will be -1, and an error message will be printed in stderr.

SEE ALSO

r2hdf(1), **a2r**(1), **dictrec**(4)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl

AUTHOR

Robert H. de Vries, Kees van der Poel, Dutch Space BV, The Netherlands.

2.1.31 r2hdf

Convert a EuroSim binary recording file into a HDF5 file

SYNOPSIS

r2hdf [**-o** *outputfile*] [**-d** *dictname*] [**-H**] [**-D**] [*recordfile*] [**-h**]

DESCRIPTION

r2hdf converts a binary EuroSim *recordfile* into the HDF5 format (see section **HDF5** for more details). This is a simple one to one translation.
The header of the input will be checked whether or not it confirms to the given dict. The base-type, size and dimensions of the dict should match the variables given in the input header. If no *recordfile* is given **r2hdf** reads from standard input.
The **r2hdf** command has the following options:

-o

The given file should be used for output instead of stdout

-d

The given dict should be used instead of the dict used in the inputfile

-H

The information from the extended header is also saved in the file, the dict is needed for the extra information.

-D

Do not use the dict found in the header of the file for converting the data.

-h

print usage message and quit.

HDF5

The Hierarchical Data Format (HDF) project involves the development and support of software and file formats for scientific data management. The HDF software includes I/O libraries and tools for analyzing, visualizing, and converting scientific data.
The HDF software is developed and supported by the National Center for Supercomputing Applications (NCSA) and is freely available. It is used world-wide in many fields, including Environmental Science, Neutron Scattering, Non-Destructive Testing, and Aerospace, to name a few. Scientific projects that use HDF include NASA's Mission to Planet Earth, and the DOE's Accelerated Strategic Computing Initiative.
HDF5 is a new, experimental version of HDF that is designed to address some of the limitations of the current version of HDF (HDF4.x) and to address current and anticipated requirements of modern systems and applications.
For more information on HDF see the NCSA HDF Home Page http://hdf.ncsa.uiuc.edu.

DIAGNOSTICS

When the program is successful, the program will have converted the given input into correct output. When the program fails, the exit status will be -1, and an error message will be printed in stderr.

SEE ALSO

r2a(1), **a2r**(1), **dictrec**(4)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl

AUTHOR

Robert H. de Vries, Kees van der Poel, Dutch Space BV, The Netherlands.

2.1.32 rec2res

Convert a EuroSim binary recording file into an Astrium RES file

SYNOPSIS

rec2res [**-o** *outputfile*] [**-d** *dictname*] [**-O** *offset*] [**-t** *col*] [**-D**] [**-m**] [**-l** *time*] [**-u** *time*] [**-c** *num-list*] [*recorder file*]
rec2res [**-h**]

DESCRIPTION

rec2res converts a binary EuroSim *recorder file* into an Astrium RES file. This is a simple one to one translation.
If no *recorder file* is given **rec2res** reads from standard input.

OPTIONS

-o *outputfile*

The given file should be used for output instead of stdout

-d *dictname*

The given dict should be used instead of the dict used in the input file

-O *offset*

Specify an offset to be added to the simulation time. This allows the user to convert the simulation time to Tref. It is assumed that the simulation time differs an integer number of seconds from Tref.

-t *col*

Specify the column containing Tref. This column will be the first column of the output file. Column counting starts at 0. By default the first column is used (simulation time).

-D

Do not use the dict found in the header of the file for converting the data.

-m

Combine multiple (split) recorder files. The assumption is made that the second recorder file name is made up of the basename of the given *recorder file* name appended with '-001' followed by the suffix of the given *recorder file* name (if any). If, for example, the first recorder file name that is passed to **rec2res** is 'counter.rec', then **rec2res** will search for counter-001.rec, counter-002.rec, and so on up to counter-999.rec. When a file is not found, **rec2res** stops processing and does not search any further for other files.

-l *time*

Specify the lower bound of the simulation time of the data to be converted. The time format is in the form of a floating point number.

-u *time*

Specify the upper bound of the simulation time of the data to be converted. The time format is in the form of a floating point number.

-c *num-list*

Specify the list of columns in the file to be converted. The list is a comma separated list of integer numbers. The column numbering starts at 0. The Tref column is added by default.

-h

print usage message and quit.

DIAGNOSTICS

When the program is successful, the program will have converted the given input into correct output. When the program fails, the exit status will be 1, and an error message will be printed on stderr.

SEE ALSO

r2a, **r2hdf**(1), **a2r**(1), **dictrec**(4)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl

AUTHOR

Robert H. de Vries, Dutch Space BV, The Netherlands.

2.1.33 replapi

Removes API headers from C or Fortran 77 source file.

SYNOPSIS

replapi [**-o** *file*] [**-a** *api*] *source-file*

DESCRIPTION

replapi removes API headers from C, Ada or Fortran 77 *source-file*.
replapi replaces the first API header found with the -if specified- file *api*. When no API-header is present, then the file *api* will be prepended. If no file *api* is specified, only the API header is removed.

OPTIONS

-o *file*

The processed *source-file* is written to *file.* If this option is omitted the processed *source-file* is written to the standard output.

-a *api*

File *api* will be inserted at the first occurence of an API-header. If '-' is specified for *api*, then contents will be read from the standard input.

DIAGNOSTICS

The following error messages can be given:

Duplicate source-file.

Only one *source-file* should specified.

Duplicate outputname option.

The option **-o** should be given at most once.

Duplicate apiname option.

The option **-a** should be given at most once.

File [*api*] to be added and stripped are identical.

Specifiy another name for *api*

File [*api*] to be added and outputname are identical.

Specifiy another name for *api*

Unknown language suffix '*x*'.

The suffix *x* is not **c**, **ads**, or **f**

Missing language suffix.

Without suffix **replapi** can't determine comment convention of the source-file.

Other diagnostics are self-explanatory, or indicate internal errors.
On error, **replapi** exits with status not 0, else 0.

SEE ALSO

api(4)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Ad Schijven, Dutch Space BV, The Netherlands.

2.1.34 schedule2sched

Convert EuroSim Mk2 schedule file to Mk3 sched file

SYNOPSIS

schedule2sched -h
schedule2sched [-v] [-m *model*] *old_schedule new_sched*

DESCRIPTION

schedule2sched converts EuroSim Mk2 schedule files to EuroSim Mk3 sched files. The Mk2 schedule file has the extension .schedule. The Mk3 schedule file has the extension .sched.
The Mk3 schedule file contains a reference to the model file from which it takes the entrypoints to schedule. This information is not present in the Mk2 schedule file as in Mk2 the schedule file was part of the model tree. The -m *model* command line option allows you to insert the reference to a model file into the new schedule file. If this is omitted the new schedule file lacks a reference to a model file and the next time you load this file in the ScheduleEditor, it will complain about the missing model file. At that point you can select the model file manually.

OPTIONS

-h

Print usage message and exit.

-v

Print verbose messages while processing the schedule file.

-m *model*

Specify the model file to use for the new schedule file.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert H. de Vries, Dutch Space BV

2.1.35 ScheduleEditor

Run the schedule editor

SYNOPSIS

ScheduleEditor [**-m** *modelfile*] [*schedulefile*]

DESCRIPTION

ScheduleEditor runs the schedule editor to view, create and/or update a schedule file. A schedule file specifies which entrypoints in the model code should be executed at which moment in time. If a *schedulefile* is passed on the command line, the schedule editor will open it.

OPTIONS

-m *modelfile*

Specify which modelfile should be used by default. The **ScheduleEditor** uses the basename of the modelfile to get the name of the data dictionary file. The data dictionary file is then opened so that the schedule editor can obtain a list of available model entrypoints.

SEE ALSO

Read the EuroSim Software User Manual (SUM) for details on how to operate the **ScheduleEditor**.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Fred van Lieshout, Dutch Space BV

2.1.36 SimulationCtrl

Run the simulation controller or observer

SYNOPSIS

SimulationCtrl [-**observer**] [-**connect** *hostname:prefcon*] [*simfile*]

DESCRIPTION

SimulationCtrl runs the simulation controller in full control mode or in observer mode when the -**observer** option is passed on the command line. The **SimulationCtrl** can be used to launch a simulator, change simulator states and terminate a simulator. If a *simfile* is passed on the command line, the SimulationCtrl will open it. A *simfile* defines which model file, schedule file and other optional files are used to run a simulator.
It is also possible to connect immediately to an already running simulator given the hostname and the connection number, using the -**connect** command line option.

OPTIONS

-observer

Run the **SimulationCtrl** in observer mode.

-connect *hostname:prefcon*

Connect immediately to the simulator running on host *hostname* on connection number *prefcon*.

SEE ALSO

Read the EuroSim Software User Manual (SUM) for details on how to operate the **SimulationCtrl**.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Fred van Lieshout, Dutch Space BV

2.1.37 subm2dict

Generate Dict from API header and source code in C or Fortran 77 source file

SYNOPSIS

subm2dict [**-o** *dict*] [**-r** *version*] [**-u** *id*] [**-l** *suffix*] [**-p** *modelpath*] [**-s***src_filename*] [**-C**] [**-D***flags*] [**–** *parser-arguments*] *source-file*

DESCRIPTION

subm2dict parses API compliant code to extract the information that is necessary to generate a Dict. The API information is stored in commented headers. **subm2dict** extracts the information in these headers, which are publicly marked data items and entry points. Format of the API header is described in **api**(4). Correct specification of data items and entry points will be checked against the source code. **subm2dict** calls **code2dict**(1) to obtain a Dict generated out of the source code. Mismatches between the information in the API header with the source code will be reported.
When **subm2dict** encounters a user defined type for a given variable, the relevant type information, will be obtained from the Dict generated by **code2dict**(1)

OPTIONS

-o *dict*

The generated Dict is written to the file *dict*. If this option is omitted the Dict is written to the standard output.

-r *version*

For future releases. Retrieve *source-file* with specified *version* from CC_engine.

-u *id*

Publicly marked data items which can be accessed in *source-file* only, eg C static variables, are prefixed in the generated Dict with *id.* It is the responsiblity of the user to provide a -unique- *id.*

-p *modelpath*

The publicly marked data items and entry points will be added with the specified *modelpath* to the hierachy of the generated Dict.

-l *suffix*

Optional *suffix* flag for specifiying language type if code is written from standard input instead of read from file. Use the *c*, *adb*, *f* suffices you'd normally use.

-s*src_filename*

Override filename of source file as it appears in the output dictionary. Useful when the source code is generated from another file. The tool merging the submodel dictionaries expects the file name of the original file, and not the one of the generated source.

C

Generate data dictionary items both for items in the API header and in the code.

-D*flags*

Debug. Specify debug flags depending on the level of detail you want. Only available if **subm2dict** was compiled with the **-DDEBUG** option.

– *parser-arguments*

Optional parser options that will be passed to **code2dict**(1).

DIAGNOSTICS

The following error messages can be given:

Duplicate outputname option.

The option **-o** should be given at most once.

File [*source-file*] to be parsed and outputname are identical.

Specifiy another outputname for *dict*

Unknown language suffix '*x*'.

The suffix *x* is not **c** or **f**

Missing language suffix.

Without suffix **api2dict** can't determine comment convention of the source-file.

syntax of another language detected

In the API header a wrong coding convention is used fi for C-code *int* var(2:3)

Error parsing API header of source-file: *source-file*

The API parser detected an error. Most likely, this message is preceded by one or more messages from the API parser.

EntryIO of variable < *variable* > conflicts with source code.

Access (**READ, WRITE, both or none**) of *variable* in source code is in conflict with the definition in API header (**Input, Output or both**). See table for legal combinations.

IN API:	PARAMETER	INPUT	OUTPUT	INOUT
IN CODE:				
NOT SPEC.	OK	OK	OK	OK
READ	OK	OK	ERROR	OK
WRITE	ERROR	ERROR	OK	OK
READWRITE	ERROR	OK	ERROR	OK

Error parsing source code of source-file: *source-file*

code2dict(1) detected an error. Most likely, this message is preceded by one or more messages from *code2dict*(1).

Merge errors : no dict will be generated.

Mismatch between the information in the API header with source code, or errors during adding information from the source code to the Dict generated from the API header. Most likely, this message is preceded by one or more comprehensive messages.

Other diagnostics are self-explanatory, or indicate internal errors.
On error, **code2dict** exits with status not 0, else 0.

ACKNOWLEDGEMENTS

The Windows version of subm2dict uses RSA Data Security, Inc. MD5 Message-Digest Algorithm.

SEE ALSO

code2dict(1), **dict**(3), **api**(4)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Ad Schijven, Dutch Space BV, The Netherlands.

2.1.38 TestAnalyzer

Run the test analyzer

SYNOPSIS

TestAnalyzer [*plotdefinitionfile*]

DESCRIPTION

TestAnalyzer runs the test analyzer. Use the test analyzer to analyze the results of a simulation run. The **TestAnalyzer** can be used to turn data from recorder files into plots (using gnuplot or PVWAVE).

SEE ALSO

Read the EuroSim Software User Manual (SUM) for details on how to operate the **TestAnalyzer**.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Fred van Lieshout, Dutch Space BV

2.2 man3: library functions

This section of manual pages contains all ‘end user library calls’ common to each EuroSim distribution. The include files for these functions are typically found in the header files of the $EFOROOT/include directory. The (shared) libraries are in $EFOROOT/lib and/or $EFOROOT/lib32.

2.2.1 esim: esimMalloc, esimFree, esimRealloc, esimCalloc, esimStrdup, esimGetSimtime, esimSetSimtime, esimGetSimtimets, esimSetSimtimets, esimGetSimtimeYMDHMSs, esimSetSimtimeYMDHMSs, esimGetWallclocktime, esimGetWallclocktimets, esimGetHighResWallclocktime, esimGetState, esimSetState, esimSetStateTimed, esimGetMainCycleTime, esimGetMainCycleBoundarySimtime, esimGetMainCycleBoundaryWallclocktime, esimGetTaskname, esimGetTaskrate, esimEntrypointFrequency, esimEntrypointGet, esimEntrypointEnable, esimEntrypointExecute, esimEntrypointFree, esimSetSpeed, esimGetSpeed, esimEventRaise, esimEventRaiseTimed, esimEventCancelTimed, esimEventCount, esimEventData, esimEventTime, esimDisableTask, esimEnableTask, esimGetRealtime, esimSetRealtime, esimGetRecordingState, esimSetRecordingState, esimVersion, esimReport, esimMessage, esimWarning, esimError, esimFatal, esimInstallErrorHandler, esimAbortNow, esimThreadCreate, esimThreadKill, esimThreadExit, esimThreadJoin, esimThreadDelete, esimSetLoadMeasureInterval, esimGetProcessorLoad, esimGetHeapUsage, esimIsResetting

EuroSim user callable functions (C interface)

SYNOPSIS

#include <esim.h>
cc ... -L$EFOROOT/lib -lesServer -les

REALTIME (SHARED) MEMORY ALLOCATION

void *esimMalloc(size_t *size*);
void esimFree(void * *ptr*);
void *esimRealloc(void * *ptr*, **size_t** *size*);
void *esimCalloc(size_t *nelem*, **size_t** *elsize*);
char *esimStrdup(const char * *str*);

REALTIME TIMING FUNCTIONS

double esimGetSimtime(void);
struct timespec esimGetSimtimets(void);
void esimGetSimtimeYMDHMSs(int *t*[7]);
double esimGetWallclocktime(void);
struct timespec esimGetWallclocktimets(void);
double esimGetHighResWallclocktime(void);
int esimSetSimtime(double *simtime*);
int esimSetSimtimets(struct timespec *simtime*);
int esimSetSimtimeYMDHMSs(int *t*[7]);

REALTIME SIMULATION STATE FUNCTIONS

esimState esimGetState(void);
int esimSetState(esimState *state*);
int esimSetStateTimed(esimState *state*, **const struct timespec *t**, **int** *use_simtime*);
struct timespec esimGetMainCycleTime(void);
struct timespec esimGetMainCycleBoundarySimtime(void);
struct timespec esimGetMainCycleBoundaryWallclocktime(void);

REALTIME TASK RELATED FUNCTIONS

const char *esimGetTaskname(void);
double esimGetTaskrate(void);
int esimEnableTask(const char *_taskname_);
int esimDisableTask(const char *_taskname_);
int esimEntrypointFrequency(esimState _state_, const char *_entrypoint_, double *_freq_);
esimEntrypoint *esimEntrypointGet(const char *_entrypoint_path_); int esimEntrypointEnable(esimEntrypoint *_entrypoint_, bool _enabled_); int esimEntrypointExecute(esimEntrypoint *_entrypoint_); int esimEntrypointFree(esimEntrypoint *_entrypoint_);
int esimGetRealtime(void);
int esimSetRealtime(int _on_);

EVENT FUNCTIONS

int esimEventRaise(const char *_eventname_, const void *_data_, int _size_);
int esimEventRaiseTimed(const char *_eventname_, const void *_data_, int _size_, const struct timespec *_t_, int _use_simtime_);
int esimEventCancelTimed(const char *_eventname_);
int esimEventData(void *_data_, int *_size_);
int esimEventTime(struct timespec *_event_occurrence_time_, struct timespec *_event_raise_time_);
int esimEventCount(const char *_eventname_);

REALTIME CLOCK FUNCTIONS

double esimGetSpeed(void);
int esimSetSpeed(double _speed_);

REALTIME RECORDING FUNCTIONS

int esimGetRecordingState(void);
int esimSetRecordingState(int _on_);

REALTIME REPORTING FUNCTIONS

void esimMessage(const char *_format_, ...);
void esimWarning(const char *_format_, ...);
void esimError(const char *_format_, ...);
void esimFatal(const char *_format_, ...);
void esimReport(int _level_, const char *_format_, ...);
int esimReportAddSeverity(const char *_sev_name_);

NON-REAL-TIME THREAD FUNCTIONS

esimThread *esimThreadCreate(const char *_name_, void (*_start_routine_)(void*), void *_arg_);
int esimThreadKill(esimThread *_thread_, int _signal_);
void esimThreadExit(int _exit_val_);
void esimThreadJoin(esimThread *_thread_);
void esimThreadDelete(esimThread *_thread_);

AUXILIARY FUNCTIONS

const char *esimVersion(void);
void esimInstallErrorHandler(ErrorHandler _user_handler_);
void esimAbortNow(void);
int esimIsResetting(void);
void main(int _argc_, char **_argv_);

METRICS FUNCTIONS

void esimSetLoadMeasureInterval(ErrorHandler *user_handler***);**
bool esimGetProcessorLoad(int *processor***, double** **avg_load***, double** **max_load***);**
void esimGetHeapUsage(int **tot_size***, int** **max_used***, int** **current_use***);**

DESCRIPTION

Required interface for simulation models that want to use the EuroSim services. These calls can be used both from three programming languages: C, Fortran and Ada (see **esim**(3C), **esim_ftn**(3) and **esim_ada**(3)).
When you link in the **libesServer.so** library a *main()* function is already included for your convenience. It makes sure all EuroSim processes are started up.
esimMalloc, **esimFree**, **esimRealloc** and **esimCalloc** are common memory allocation functions. These are the same as their **malloc**(3) counterparts in the "C" library, with the exception that the EuroSim calls are optimised for parallel/realtime usage, and checks for memory exhaustion are built-in. For the semantics and arguments and return values see **malloc**(3) for details.
esimGetSimtime() returns the simulation time in seconds with the precision of the basic period with which the simulation runs (5 ms by default). In case the simulation is driven by the external interrupt the precision is equal to that period. If the simulator has real-time errors the simulation time will be slower than the wall clock. The simulation time is set to zero (0) on arriving in initialising state. **esimGetSimtimets**() returns the simulation time in a timespec structure. **esimGetSimtimeYMDHMSs**() returns the simulation time in an array of 7 int's containing: year, month, day, hour, minute, second and nanoseconds.
esimSetSimtime() sets the requested simulation time *simtime* in seconds. This can only be done in the standby state. If calling **esimSetSimtime** in any other state is attempted or *simtime* is less than zero, no simulation time is set and (-1) is returned. On success zero (0) is returned. **esimSetSimtimets**() sets the simulation time using a timespec structure. **esimSetSimtimeYMDHMSs**() set the simulation time using an array of 7 int's containing: year, month, day, hour, minute, second and nanoseconds.
esimGetWallclocktime() returns the wall clock time in seconds. The basic resolution is equal to the resolution of the high-res time described next, but it is truncated to milliseconds. **esimGetWallclocktimets**() returns the wall clock time in a timespec structure. The wall clock time is set to zero (0) when the first model task is scheduled, and runs real-time, which means that it is independent from the simulation time.
esimGetSimtimeUTC(), **esimSetSimtimeUTC**() and **esimGetWallclocktimeUTC**() are obsolescent. Use **esimGetSimtimets**(), **esimSetSimtimets**() and **esimGetWallclocktimets**() instead.
If your models need absolute times instead of relative times, you can change the support option in the ModelEditor, using the Tools | Set Build Options menu command.
esimGetHighResWallclocktime() returns the "same" time as **esimGetWallclocktime** but in milliseconds and with a higher resolution. This high resolution is 21 ns on high-end platforms equipped with a hardware timer such as a challenge and onyx. On low end platforms this resolution as can be achieved by the **gettimeofday**(3B) call. For more information on resolutions of this call see **timers**(5).
esimGetState() returns the current simulator state. The state can be any of the following values: *esimUnconfiguredState*, *esimInitialisingState*, *esimExecutingState*, *esimStandbyState* or *esimStoppingState*.
esimSetState() sets the simulator state to the indicated value *state*. *state* can be any of the following values: *esimUnconfiguredState*, *esimInitialisingState*, *esimExecutingState*, *esimStandbyState* or *esimStoppingState*. If *state* is not reachable from the current state *0* is returned; on a successful state transition *1* is returned.
esimSetStateTimed() sets the simulator state to the indicated value *state* at the specified time *t*. The possible values of *state* are listed in the previous paragraph. If the flag *use_simtime* is set to 1 (true), the specified time is interpreted as simulation time. If the flag is set to 0 (false), the specified time is interpreted as the wallclock time. The transition time uses a struct timespec where the number of seconds is relative to January 1, 1970. On success this function returns 0, otherwise -1.

esimGetMainCycleTime() returns the main cycle time of the schedule. The result can be used to compute valid state transition times for use in the function esimSetStateTimed().
esimGetMainCycleBoundarySimtime() returns the simulation time of the last state transition. This boundary time can be used to compute valid state transition times for use in the function esimSetStateTimed() when the value of use_simtime is true.
esimGetMainCycleBoundaryWallclocktime() returns the wallclock time of the last state transition. This boundary time can be used to compute valid state transition times for use in the function esimSetStateTimed() when the value of use_simtime is false.
esimGetTaskname() returns the name of your current task.
esimGetTaskrate() returns the frequency (in Hz) of your current task.
esimEntrypointFrequency() stores the frequency (in Hz) of the entrypoint with the name *entrypoint* in the argument *freq* in the state *state*. If the entrypoint appears multiple times in the schedule the function returns -1. If the entrypoint does not appear in the schedule in the given state, the frequency is 0.
esimEntrypointGet() looks up the entry point given by the data dictionary path *entrypoint_path*. If the entry point cannot be found, the function will return a NULL pointer. The pointer must be freed after use by calling **esimEntrypointFree**().
esimEntrypointEnable() enables or disables an entry point. This means that the entry point does not get executed in the schedule or via MDL commands or via **esimEntrypointExecute**(). The entry point handle *entrypoint*, must be looked up first using **esimEntrypointGet**(). If the *enabled* flag is true, the entry point is enabled for execution, if the flag is false, the entry point is disabled.
esimEntrypointExecute() executes an entry point. The entry point handle *entrypoint* must be looked up first using **esimEntrypointGet**().
esimEntrypointFree() is used to free the memory allocated by **esimEntrypointGet**().
esimDisableTask() disables the task *taskname* defined with the Schedule Editor **ScheduleEditor**(1). It will be skipped (not executed) by the EuroSim runtime until a call is made to **esimEnableTask**.
esimEnableTask() enables the task *taskname* defined with the Schedule Editor **ScheduleEditor**(1). It will be executed/scheduled according to the schedule made with the Schedule Editor.
esimGetRealtime() returns the current operational state of the EuroSim realtime Scheduler. If *1* is returned, hard realtime execution is in progress, whereas a return of *0* indicates that your model is not executing in realtime mode.
esimSetRealtime() sets the current operational state of the EuroSim realtime Scheduler. Hard real time execution can only be set if the scheduler was launched in hard real time mode. *1* is returned on success. *0* is returned on failure.
esimEventRaise() raises the event *eventname* for triggering tasks defined with the Schedule Editor **ScheduleEditor**(1). User defined data can be passed in *data* and *size*. On success this function returns 0, otherwise -1.
esimEventRaiseTimed() raises the event *eventname* for triggering tasks defined with the Schedule Editor **ScheduleEditor**(1) at the specified time *t*. User defined data can be passed in *data* and *size*. If the flag *use_simtime* is set to 1 (true), the specified time is interpreted as simulation time. If the flag is set to 0 (false), the specified time is interpreted as the wallclock time. The transition time uses a struct timespec where the number of seconds is relative to January 1, 1970. On success this function returns 0, otherwise -1.
esimEventCancelTimed() cancels pending event(s) *eventname* that were scheduled by a call to **esimEventRaiseTimed** On success this function returns 0, otherwise -1.
esimEventCount() returns the number of times that event *eventname* has been raised, or -1 if no such event is defined.
esimEventData() returns the event data that was passed with the event that activated the input connector connected to the current task. The input value of *size* specifies the size of the *data* buffer. The output value of *size* is the size of the retrieved data. On success this function returns 0, otherwise -1.
esimEventTime() returns the timestamp on which EuroSim detected the occurrence of the event, and the timestamp on which it injected the event into the realtime domain via an input connector. This function must be called from a task that is directory connected to an input connector. On success this function returns 0, otherwise -1.

esimGetSpeed() returns the current speed of EuroSim Scheduler. e.g. *1.0* means (hard or soft) real time. *0.1* means slowdown by a factor 10. *-1* means as fast as possible.
esimSetSpeed() sets the current speed of EuroSim Scheduler. e.g. *1.0* means (hard or soft) real time. *0.1* means slowdown by a factor 10. *-1* means as fast as possible. The speed can only be changed if the scheduler is running non realtime. If *speed* is not a feasible speed *0* is returned; on a successful setting of the speed *1* is returned.
esimGetRecordingState() returns the current state of the EuroSim realtime data Recorder. If *1* is returned, data is logged to disk, whereas a return of *0* indicates that recording is switched off.
esimSetRecordingState() sets the state of the Recorder to *on*. If *on* is *1* data will subsequently be written to disk, if *on* is *0* data recording will be suspended. Return value is either *1* or *0*, depending on success or not.
The functions **esimReport**, **esimMessage**, **esimWarning**, **esimError** and **esimFatal** can be used to send messages from the EuroSim Model code to the test-conductor interface. The **esimReport** function allows the caller to specify the severity of the message. The other functions have implicit severities. The possible severity levels are:

```
esimSevMessage  comment or verbose information
esimSevWarning  warning messages
esimSevError    error messages
esimSevFatal    non-recoverable errors
```

In the C interface routines the message consists of a format string *format* and its optional arguments. (see **printf**(3)). In the Fortran interface routines the message consists of a single string argument *message*.
esimReportAddSeverity() creates a new severity string with a user-defined name to be used in **esimReport**().
esimThreadCreate() creates a non-real-time thread in the same address space as the rest of the simulator. The name of the thread is given in *name*. The new thread starts executing *start_routine* with argument *arg*.
esimThreadKill() sends signal *signal* to thread *thread*.
esimThreadExit() exits the currently running thread. The return code is in *exit_val*.
esimThreadJoin() waits for the thread *thread* to finish.
esimThreadDelete() frees all memory of thread *thread*. The thread must have exited before this call can be performed. This means that you must first call **esimThreadJoin**() before calling **esimThreadDelete**().
esimVersion() returns a string indicating the current version of EuroSim that you are running.
esimInstallErrorHandler() installs a user-defined error handler callback of the form:

```
void user_handler(esimErrorScope scope,
                  const char *object_id)
```

This callback function is called when an error occurs that may need intervention in user code. If the *scope* is **esimDeadlineError** then the *object_id* is the name of a task in the simulator schedule that has exceeded its deadline by a factor of ten. This allows a model developer to take action (f.i. force a core dump) when part of a model is ill-behaved (never ending loops). If no error handler is installed, the default action is to disable the offending task and enter the stand-by state. Note that deadline checking is only performed in real-time mode.
Passing a NULL pointer will de-install the user error handler. No stack of user error handlers is maintained.
esimAbortNow() immediately starts termination and cleanup of the simulator. This is useful when an error condition is found (f.i. at initialisation time) and no more entrypoints should be scheduled for execution.
esimIsResetting() returns 1 when the reset procedure is in progress and 0 when it is not. The reset procedure starts in standby state and progresses through exiting, unconfigured, initializing and back into standby state. This function allows you to distinguish between for example a user initiated state transitions to exiting state to stop the simulator and the state transitions performed in the reset procedure.

esimGetHeapUsage() returns the total size of the heap, the maximum amount used since startup and the current use (all in bytes).
esimSetLoadMeasureInterval() sets the time interval (msec) over which the processor load percentages will be measured. Intervals will be synchronized to the basic cycle such that: interval % basic_cycle_no = 0 Synchronisation to the basic cycle and load measurements are started at the end of the running load interval (if none was set by a previous call to this function, synchronisation is started immediately) The function returns false when it fails. Otherwise true.
esimGetProcessorLoad() reads the maximum and average load (percentage) of the specified processor. The maximum percentage is defined as the maximum percentage of time a processor was executing esim tasks during the time interval set by esimSetLoadInterval. The maximum is reset after each call to esimSetLoadInterval. The average load percentage is defined as the average of the load percentages measured the number of times the time intervals passed since the last call to this function. I.e. if the time interval is set to the main cycle period (by esimSetLoadInterval)and this function is called every fourth main cycle then the average load is calculated over the load of the last four (completed) main cycles. The function returns false when it fails. Otherwise true.
main() is the main entry point for a simulator made with the **libesServer.so** library. and does the parsing of any command line options passed to the simulator. The command line options are listed in the **OPTIONS** section and are included for completeness. As in the EuroSim environment starting up simulators will be done through a special daemon (See **esimd**(1M)) you need not worry about these options.

OPTIONS

The following are the command line options that can be passed to the *main()* function of the simulator:

-v

enable verbose printing of currently running simulator.

***-u* uid**

(numeric) uid for file ownership of recordings etc.

***-g* uid**

(numeric) gid for file ownership of recordings etc.

***-c* number**

connection number offset for the simulator process. Needed if more simulators are to be run on one host.

***-x* simfile.sim**

Sim file to load.

***-m* missionfile.mdl**

mission to initially load.

***-M* modelfile.model**

model file.

***-i* initialcondfile**

initial condition file to load.

***-d* datadict.dict**

data dictionary to load.

***-s* schedule.sched**

schedule file to load the Scheduler with.

***-e* exportsfile.exports**

exports file to use for external simulator access.

***-f* number**

frequency to run the asynchronous processes with. The default for the asynchronous frequency is 2 Hz.

***-R* dir**

directory to write recording files to. Defaults to the directory where the *missionfile.mdl* file came from.

***-l* number**

period (with asynchronous frequency) for data logger. Every *number*'th cycle data values will be delivered to (interested) clients (e.g. a scenario manager with a data monitor. Defaults to 1.

***-r* number**

realtime flag. When number is 0 realtime mode is off, when it is 1 it is on.

-E

do not use the daemon for services (such as CPU allocation).

-I

do not go to initializing state automatically.

-Dflags

Debugging flags. Only available when EuroSim libraries were compiled with DEBUG defined.

-S

Stand-alone mode (do not wait for client to connect). Useful for debugging. Use in combination with **-E** option.

NOTES

libesServer.so contains all EuroSim realtime libraries including those from *ActionMgr* (see **actionmgr**(3)), *Scheduler* (see **Sched_***(3)), *Recorder* and *Stimulator* (see **recstim**(3)), *Script* (see **script**(3)), etcetera.....

SEE ALSO

esimd(1), **printf**(3)

WARNINGS

The message handling routines use ring buffers to decouple the realtime and non realtime domains of EuroSim. The size of these ring buffers and therefore the maximum message size is limited (currently 20*BUFSIZ as defined in **stdio.h**). Messages may be lost if the message rate is too high for the message handler.

BUGS

We'll need a **esimSetWallClockTime**, **esimGetTaskoffset** and *esimSetTaskoffset* one day.
A non-real-time non-parallel version of this library should be made.
One day we should get rid of *main()* in this library.
Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Wim Elhorst, Marc Neefs, Robert de Vries and Fred van Lieshout for Dutch Space BV, The Netherlands.

2.2.2 esimCalibrationLookup, esimCalibrate, esimCalibrationErrorString

Calibration routines

SYNOPSIS

#include <esimCalibration.h>
const EsimCalibration_t *esimCalibrationLookup(const char *_name_);
EsimCalStatus_t esimCalibrate(const EsimCalibration_t *_cal_, double _in_, double *_out_);
const char *esimCalibrationErrorString(EsimCalStatus_t _stat_);

DESCRIPTION

esimCalibrationLookup looks up the calibration curve with the given name. The returned value can then be used in **esimCalibrate** as parameter *cal* to perform the actual calibration. The input value *in* is calibrated into the output value *out*. The return code of the function **esimCalibrate** indicates success or failure of the calibration.
The return code can be converted to an error string with the function **esimCalibrationErrorString**. The parameter *stat* is the return code of the function **esimCalibrate**.

DIAGNOSTICS

esimCalibrationLookup will return NULL if the calibration curve with the given name does not exist.
esimCalibrationErrorString always returns an error string. If you pass it an unknown status value, the string contains "Unknown error occurred during calibration".
esimCalibrate may return the following return codes:

ESIM_CAL_OK

Calibration succeeded, no error.

ESIM_CAL_LOOKUP_FAILED

Lookup calibration failed, lookup value was not in the lookup table.

ESIM_CAL_INPUT_TOO_SMALL

Input value below allowed minimum.

ESIM_CAL_INPUT_TOO_LARGE

Input value above allowed maximum.

SEE ALSE

CalibrationEditor(1)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert de Vries, Dutch Space BV

2.2.3 esimEHInstall, esimEHForward, esimEHDispatch

External event handler functions

SYNOPSIS

#include <esim.h>
#include <esimEH.h>
typedef int (*esimEH_Associate)(const char *name, void *user_data);
typedef void (*esimEH_Dissociate)(const char *name, int id, void *user_data);
typedef int (*esimEH_IntrHandler)(esimEH *context, void* user_data);
typedef int (*esimEH_Dispatcher)(esimEH *context, void* user_data, const void* msg, int size);
int esimEHDispatch(esimEH *context, int id, const void *msg, int size);
int esimEHForward(esimEH *context, const void *msg, int size);
int esimEHInstall(const char *name, esimEH_Associate associate, esimEH_Dissociate dissociate, esimEH_IntrHandler intr_handler, esimEH_Dispatcher dispatcher, void *user_data);
int esimEHUninstall(const char *name);

DESCRIPTION

The External Event Handler **esimEH** provides a general interface to handle interrupts, signals and synchronisation with named semaphores in EuroSim models.
The External Event Handler must have been created with the Schedule Editor. Creating an external event handler with the default dispatcher causes automatic creation of an input connector menu item in the Schedule Editor. This input connector has (must have) the same name as the event handler.
Only external event handlers that do not use the default dispatcher have to use the functions described below.
The External Event Handler has its own execution thread. During installation with **esimEHInstall** this thread calls the **associate** function to get unique id's for every event. After that it calls the **dispatcher** function every time an external event arrives. Events with data can be passed from the handler to an input connector using **esimEHDispatch** with the event id. In a task connected directly to the input connector the event data can be retrieved with **esimEventData**. The time at which EuroSim became aware of the external event and the time at which it injected the event into the scheduler can be retreived with **esimEventTime**.
An interrupt handler (**intr_handler**) can be installed if the external event handler is connected to a HW interrupt. From the interrupt handler the user can forward events with data to the dispatcher using **esimEHForward**. The interrupt handler could be necessary if the hardware has to be accessed in interrupt context to read status and clear the interrupt. Note that the interrupt context has several restrictions (see below).

ACCESS ROUTINES

esimEHInstall

installs callback functions for external event handler *name*. In the esimEHInstall call the *associate* function is called for every event connected to the external event handler. This function should return a unique id >= 0 for every event. The *dissociate* function is called for every associated event when the simulator terminates or when the event handler is uninstalled. *intr_handler* is called for every interrupt. This function can only be used in combination with a HW interrupt source, otherwise it should be NULL. From the interrupt handler the dispatcher can be activated with data using **esimEHForward**. If the interrupt handler is not installed the dispatcher will be called every time an external event arrives. From the *dispatcher* input connectors can be activated using **esimEHDispatch** with the id of the event. *user_data* is passed to all callback functions. *dissociate* and *user_data* can be NULL. On success this function returns 0, otherwise -1.

esimEHUninstall

uninstalls the previously installed callback functions. In the esimEHUninstall call the *dissociate* function is called for every associated event.

esimEHForward

passes the event with *msg* of *size* to the dispatcher. *context* should be the context parameter of the callback function. On success this function returns 0, otherwise -1.

esimEHDispatch

passes the event with *id* and *msg* of *size* to an input connector. This data can be retrieved with **esimEventData** see **esim**(3). *context* should be the context parameter of the callback function. On success this function returns 0, otherwise -1.

CALLBACK FUNCTIONS

esimEH_Associate

is called by esimEHInstall for every event connected to the external event handler. A unique id >= 0 for the event with *name* should be returned. If id *esimEH_NOT_ASSOCIATED* is returned the association of *name* is not performed and the event connector with that name will never be activated. This function can also be used to do any specific initialisations for the event. *user_data* is the pointer that was passed to esimEHInstall.

esimEH_Dissociate

is called during termination of the simulator for every event connected to the external event handler. This function can be used to do any specific termination action for the event with *name* and *id*. *user_data* is the pointer that was passed to esimEHInstall.

esimEH_IntrHandler

is called for every interrupt of the external event. This callback can be used to read HW status and clear the interrupt. *context* should be passed to any call to esimEHForward. *user_data* is the pointer that was passed to esimEHInstall. On success this function should return 0, otherwise -1.

esimEH_Dispatcher

If the event handler has an interrupt handler it will be called only if esimEHForward has been called. The data from esimEHForward is passed in *msg* and *size*. If the event handler does not have an interrupt handler msg and size are NULL and 0, respectively. *context* should be used in esimEHDispatch. *user_data* is the pointer that was passed to esimEHInstall. On success this function should return 0, otherwise -1.

NOTES

Events are dispatched by id and not by name for performance reasons.
Event data is restricted in size to 256 bytes

WARNINGS

REAL TIME PERFORMANCE

External event handlers interrupt the real-time scheduler, and thus influence the real-time performance of the system. To prevent jitter on the scheduler clock the external event handlers should be installed on other processors than the clock. Best preformance can be obtained if the processor with the external event handler does not run any periodic tasks.

RESTRICTIONS ON DISPATCHER CALLBACK

Only the following functions can be used from **esim**(3):

- realtime (shared) memory allocation functions
- realtime timing functions
- realtime simulation state functions
- realtime reporting functions, however they do not raise the NOTICE, WARNING, ERROR and FATAL events.

INTERRUPT HANDLERS

On SGI 6.5 the interrupt handlers are ULI's, see **uli**(3), which have strong restrictions that cause interrupt handler crashes due to an illegal instruction. Only the following functions can be used from **esim**(3):

- esimGetSimtime
- esimGetWallclocktime
- esimGetState

SEE ALSO

esimEI(3), **uli**(3)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Sander de Snoo, Origin Copyright EuroSim Consortium 2001

2.2.4 esimEIOpen, esimEIClose, esimEIInstall, esimEIPulse, esimEIPoll, esimEIBusywait, esimEIClear

General operations for external interrupts, built on "/dev/ei".

SYNOPSIS

#include <esim.h>
#include <esimEI.h>
esimEI_t esimEIOpen(const char *filename);
bool esimEIClose(esimEI_t *ei);
bool esimEIInstall(esimEI_t *ei, int signal, esimEI_Handler handler);
int esimEIFd(esimEI_t *ei);
bool esimEIPulse(esimEI_t *ei, int connector);
bool esimEIPoll(esimEI_t *ei);
bool esimEIBusywait(esimEI_t *ei);
void esimEIClear(esimEI_t *ei);

DESCRIPTION

The External Interrupt unit *esimEI* provides a general interface to access external interrupts on a SGI Challenge/Onyx L/XL /Origin 200/2000. The current implementation is built on the Irix 6.5 external interrupt device driver. The esimEI unit provides the following services:

- EuroSim tasks can generate output interrupts to one of the SGI output connectors.
- A user can install a user defined handler for incoming interrupts.
- Input interrupts can be used as the (external) real-time clock of the scheduler, instead of the default itimer mechanism (see also DDD and Sched_Clock unit).

ACCESS ROUTINES

esimEIOpen

Opens the device with the indicated filename (e.g. /dev/external_int/1). esimEIOpen returns TRUE iff. the device is opened, FALSE otherwise.

esimEIClose

Closes the external interrupt device. esimEIClose returns TRUE iff. the device is closed, FALSE otherwise.

esimEIInstall

Installs a handler for incoming interrupts on the external interrupt device. The signal that will be send to the calling process is specified in *signal*, the handler is specified by the function pointer *handler*. esimEIInstall returns TRUE iff. the handler is installed, FALSE otherwise.

esimEIFd

Returns the file descriptor of the opened external interrupt device.

esimEIPulse

Generate one output interrupt on the specified output connector(s) of the external interrupt device. With bit 0..3 of *connector*, it is specified on which output connector(s) the interrupt should occur (simultaneous). esimEIPulse returns TRUE iff. the output interrupt(s) are generated successfully, FALSE otherwise.

esimEIPoll

Check for outstanding interrupts for the external interrupt device, and dequeue one of the outstanding interrupts. esimEIPoll returns TRUE iff. there are outstanding interrupts, FALSE otherwise.

esimEIBusywait

Wait until an interrupt occurs on the external interrupt device, and dequeue one of the the outstanding interrupts. If there are already outstanding interrupts, this function will return directly without waiting. esimEIBusywait returns TRUE iff. an interrupt has occurred, FALSE otherwise.

esimEIClear

Clear all outstanding interrupts for the external interrupt device.

WARNINGS

User defined interrupt handlers

If ”direct” response is required on interrupt occurrence, the user may install a user defined interrupt handler, however, the following warnings should be taken into account. 1) When interrupts will occur very fast after each other, some signals can be lost (SYSV Unix anomaly) so that not all interrupts will be handled. 2) This mechanism interrupts the real-time scheduler, and thus the real-time performance and/or (locking) behavior of the system. Be careful with this mechanism!

External real-time clock

When the scheduler is driven by the external interrupts, the user cannot install a user defined interrupt handler, because it will overwrite the existing scheduler ”heartbeat” handler, which causes scheduler hangup.

SEE ALSO

Sched_Clock(3) **esimEH**(3)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Johannes Bosman, Origin/Niewegein BV, The Netherlands.

2.2.5 esimErrInj: userErrInjPublish, esimErrInjPublishParameter, esimErrInjFunct_t, esimErrInjPublishFunction, esimErrInjSetPostFix, esimErrInjGetBooleanValue, esimErrInjGetIntegerValue, esimErrInjGetUnsignedIntegerValue, esimErrInjGetDoubleValue

EuroSim error injection functions

SYNOPSIS

#include <esimErrInj.h>

Publication Functions

int userErrInjPublish(esimErrInjPublish_t *_pub_);
int esimErrInjPublishParameter(esimErrInjPublish_t *_pub_, const char *_name_, const char *_description_, const char *_unit_, esimErrInjDataValue_t _value_, bool _internal_state_);
typedef esimErrInjDataValue_t (*esimErrInjFunc_t)(esimErrInj_t *_error_, esimErrInjDataValue_t _input_);
void esimErrInjPublishFunction(esimErrInjPublish_t *_pub_, esimErrInjFunc_t _func_);
int esimErrInjSetPostFix(const char *_postfix_);

Runtime Functions

int esimErrInjGetBooleanValue(esimErrInj_t *_error_, int _parameter_id_, bool **_value_);
int esimErrInjGetIntegerValue(esimErrInj_t *_error_, int _parameter_id_, int **_value_);
int esimErrInjGetUnsignedIntegerValue(esimErrInj_t *_error_, int _parameter_id_, unsigned int **_value_);
int esimErrInjGetDoubleValue(esimErrInj_t *_error_, int _parameter_id_, double **_value_);

DESCRIPTION

The publication functions are used to publish the error injection function to EuroSim. The user must implement the publication function *userErrInjPublish* and the error injection function of type *esimErrInjFunc_t*.
The publication function publishes the parameters used by the error injection function, a reference to the error injection function itself and a post-fix to be used for the error injection parameters.
The function *userErrInjPublish* is called during the build process and during initialization of the simulator. During the build process the information regarding the error injection parameters is used to construct the EuroSim data dictionary. The parameter *pub* must be passed to the publication functions.
The function *esimErrInjPublishParameters* is used to publish the parameters used by the error injection function. The function returns an id that must be saved for later use by the run-time functions to retrieve the parameter values, or -1 on error. The *pub* parameter must be the *pub* parameter passed to the *userErrInjPublish* function. The *name* parameter is the name of the error injection parameter. The *description* parameter is the description of the error injection parameter. The *unit* parameter allows you to specify the physical unit of the parameter. The *value* parameter is used to specify both the type and the initial value of the error injection parameter. The *internal_state* flag is used to indicate that the error injection parameter is an internal state parameter that is hidden somewhat from the user in a sub-structure in the data dictionary. This is to be used for parameters that should not be modified by end-users as they store the internal state of the error injection function.
The function *esimErrInjPublishFunction* is used to publish the error injection function itself. The *pub* parameter must be the *pub* parameter passed to the *userErrInjPublish* function. The *func* parameter is the error injection function.
The function *esimErrInjSetPostFix* is used to set the post-fix of the error injection variables generated in the data dictionary. The default post-fix is "_error". This can be changed to anything except the empty string. The function returns 0 on success, -1 on failure.

The run-time functions are used to retrieve the error injection function parameters. The error injection function receives an *error* parameter that refers to an instance of the error injection parameters and an *input* parameter that contains the input value. The return value is the output value. The *error* parameter must be passed to the *esimErrInjGet*Value* functions as the *error* argument. The error injection function must first retrieve the error injection parameter values using the *error* parameter and the parameter *id*. Depending on the type of the input value the function may implement different error injection strategies. The error injection parameters are retrieved as pointers to the actual values so that the error injection function can modify them. The return value must have the same type as the input value and must contain the (optionally) modified input value.
The functions *esimErrInjGetBooleanValue*, *esimErrInjGetIntegerValue*, *esimErrInjGetUnsignedIntegerValue* and *esimErrInjGetDoubleValue* are used to retrieve the pointers to parameter values. The function returns -1 if the *parameter_id* is illegal or in case of a type mismatch between the *parameter_id* and the function. The *error* parameter is the handle to the error injection parameter instance passed to the error injection function as the *error* argument. The *parameter_id* parameter is the id of the parameter as returned by the *esimErrInjPublishParameter* function. The *value* parameter is used to retrieve the pointer to the parameter value.

DIAGNOSTICS

Functions that return an integer use -1 to indicate an error. Positive values or 0 indicate success.

SEE ALSO

EuroSim Software User Manual Appendix on Error Injection.

BUGS

No known bugs yet.
Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert de Vries for Dutch Space BV, The Netherlands.

2.2.6 esimMil1553Open, esimMil1553Close, esimMil1553SetMode, esimMil1553Start, esimMil1553Stop, esimMil1553Poll, esimMil1553BcAdd, esimMil1553BcRead, esimMil1553BcGetDbufHdr, esimMil1553BcWrite, esimMil1553RtAdd, esimMil1553RtRead, esimMil1553RtGetDbufHdr, esimMil1553RtWrite, esimMil1553BmAdd, esimMil1553BmReadFirst, esimMil1553BmReadNext, esimMil1553SetMajor, esimMil1553SetMinor, esimMil1553RtSetRegister, esimMil1553SetXioRegister, esimMil1553GetXioRegister, esimMil1553WriteMrtmonRa esimMil1553ReadMrtmonRam, esimMil1553Selftest, esimMil1553BcGetCb, esimMil1553RtGetCb, esimMil1553BcCbIter

General operations for MIL1553 Bus control, built on "VMIVME-6000 BCU software".

SYNOPSIS

#include <esimMil1553.h>
int esimMil1553Open(int unit, const char* A16_name, const char *A24_name);
int esimMil1553Close(int unit);
int esimMil1553SetMode(int unit, int mode);
int esimMil1553Start(int unit);
int esimMil1553Stop(int unit);
int esimMil1553Poll(int unit);
int esimMil1553BcAdd(int unit, int rt, int sa,int type, int len, ...);
int esimMil1553BcRead(int unit, int rt, int sa,uword *buf, int len);
int esimMil1553BcGetDbufHdr(int unit, int rt, int sa,uword tf_flag, bc_dbuf_hdr_t *hdr);
int esimMil1553BcWrite(int unit, int rt, int sa, uword *buf, int len);
int esimMil1553RtAdd(int unit, int rt);
int esimMil1553RtRead(int unit, int rt, int sa, uword *buf, int len);
int esimMil1553RtGetDbufHdr(int unit, int rt, int sa, rt_rw_mode rw_mode, rt_dbuf_hdr_t *hdr);
int esimMil1553RtWrite(int unit, int rt, int sa, uword *buf, int len);
int esimMil1553BmAdd(int unit, int rt);
int esimMil1553BmReadFirst(int unit, struct MON_db *data, mon_rb_cursor_t *cursor);
int esimMil1553BmReadNext(int unit, struct MON_db *data, mon_rb_cursor_t *cursor);
int esimMil1553SetMajor(int unit, long frame);
int esimMil1553SetMinor(int unit, long frame);
int esimMil1553RtSetRegister(int unit, int name, int value);
int esimMil1553SetXioRegister(int unit, int name, int value);
int esimMil1553GetXioRegister(int unit, int name, int *value);
int esimMil1553WriteMrtmonRam(int unit, int addr, int value);
int esimMil1553ReadMrtmonRam(int unit, int addr, int *value);
int esimMil1553Selftest(int unit, int *asr);
volatile struct BC_CB* esimMil1553BcGetCb(int unit, int rt, int sa, int tf_flag);
volatile struct RTMON_CCB* esimMil1553RtGetCc(int unit, int rt);
typedef void (*bc_iter_func_t)(int unit, MEMPOINTER(struct BC_CB) bc_cb, void *user_data);
void esimMil1553BcCbIter(int unit, bc_iter_func_t callback, void *userdata);

DESCRIPTION

The MIL1553 unit *esimMil1553* provides a general interface to access a MIL1553 bus. The current implementation is built on the VMIVME-6000 BCU software driver library.

ACCESS ROUTINES

esimMil1553Open

Open and initialise the MIL1553 device. The *unit* should be a number in the range 0-4. This number is returned if the device is successfully opened. On failure -1 is returned. The jumper settings and associated VME memory ranges of the VMIC VME6000 board for each device is as in the table below. Each board has a memory range in the A24 space of 0x40000 bytes (256 KB). A zero (0) in the jumper setting column means a removed jumper. For more information on jumper settings see section 5-3 on pages 5-1 to 5-6 of the VMIC manual.

unit	jumper settings E1-E8	IO range (A16)	Memory range (A24)
0	11111111	0xC000	0x800000
1	11111110	0xC040	0x840000
2	11111101	0xC080	0x880000
3	11111100	0xC0C0	0x8C0000
4	11111011	0xC100	0x900000

The *A16_name* and *A24_name* parameters are the VME device path names for the A16 and A24 address space. The following table shows the device names for some common systems. Check the hardware owner's manual if your system is not listed.

system	VME interface	A16 device path	A24 device p
SGI Challenge	built-in	/dev/vme/vme21a16n	/dev/vme/vme
Origin 200	XIO/VME bridge	/hw/vme/1/usrvme/a16n/d16	/hw/vme/1/us
Origin 200	Bit3 PCI/VME bridge	/hw/bit3/965/unit0/bt_axsri	/hw/bit3/965

esimMil1553Close

Close the MIL1553 device that is specified by *unit*. esimMil1553Close returns 0 iff. the MIL153 device is closed, -1 otherwise.

esimMil1553SetMode

Set the operation mode of the MIL1553 device *unit*. The operation mode is specified by *mode* and can be one of the following: **MODE_BCSIM** for bus controller mode, **MODE_MRTMON** for remote terminal and bus monitor mode. esimMil1553SetMode returns 0 iff. the mode is set, -1 otherwise.

esimMil1553Start

Start the MIL1553 device *unit* with the execution of its scenario (control blocks). In **MODE_BCSIM** the device will stop after execution of the scenario. In **MODE_MRTMON** the device will be in execute state until the esimMil1553Stop will be invoked. The programmer must call esimMil1553Poll() to check whether the scenario has finished running before another call to this function can be done. esimMil1553Start returns 0 iff. the device is started, -1 otherwise.

esimMil1553Stop

Stop the MIL1553 device *unit* from the execute state in the modes **MODE_MRTMON**. esimMil1553Stop returns 0 iff. the device is stopped, -1 otherwise.

esimMil1553Poll

Check if the MIL1553 device *unit* is still executing. esimMil1553Poll returns 0 iff. the device is still busy, 1 iff. the device is ready, -1 otherwise. The programmer must call this function at least once after each esimMil1553Start() call to test whether the scenario has finished running and to reset the internal driver state for the next run.

esimMil1553BcAdd

Append a transfer action (control block) to the MIL1553 device *unit* which must be in the **MODE_BCSIM** mode. The remote terminal is specified by *rt* and the sub-address by *sa* for *type* is **RT_BC_TRANSFER** or **BC_RT_TRANSFER**. The length of the transfer (*len*) is in words (16 bits). For **RT_RT_TRANSFER** two extra int parameters need to be supplied to specify the transmitting remote terminal and its sub-address. The first extra integer argument is the remote terminal address and the second extra integer argument is the sub-address. The standard *rt* and *sa* parameters specify the receiving remote terminal and its sub-address. esimMil1553BcAdd returns 0 iff. the transfer action (control block) is added to the scenario, -1 otherwise.

esimMil1553BcRead

Read data of the last transfers from the MIL1553 device *unit* and in the **MODE_BCSIM** mode. The remote terminal *rt* and sub-address *sa* specify which read buffer must be copied to *buf* from the scenario (control blocks). The size of the copy *en* is in words (16 bits). esimMil1553BcRead returns 0 iff. the read is succeeded, -1 otherwise.

esimMil1553BcGetDbufHdr

Read the values of the Interrupt Status Register, Transmission Status Register, Status Word 1 and Status Word 2 from the bus controller control block where these values are posted after each transfer. These values can be used to check whether the transfer was successful.

esimMil1553BcWrite

Write data to the transfer buffers of the MIL1553 device *unit* and in the **MODE_BCSIM** mode. The remote terminal *rt* and sub-address *sa* specify which write buffer must be written with the contents of *buf*. The size of the copy *len* is in words (16 bits). esimMil1553BcWrite returns 0 iff. the write is succeeded, -1 otherwise.

esimMil1553RtAdd

Append a remote terminal buffer (control block) to the MIL1553 device *unit* and in the **MODE_MRTMON** mode. The remote terminal is specified by *rt*. esimMil1553RtAdd returns 0 iff. the remote terminal is appended to the scenario (control blocks), -1 otherwise.

esimMil1553RtRead

Read data of the last RT transfer from the MIL1553 device *unit* and in the **MODE_MRTMON** mode. The remote terminal *rt* and sub-address *sa* specify which read buffer must be copied to *buf* from the scenario (control blocks). The size of the copy *len* is in words (16 bits). esimMil1553RtRead returns 0 iff. the read is succeeded, -1 otherwise. If 0 is passed as value for parameter *len* the actual number of words to be read is derived from the command word.

esimMil1553RtGetDbufHdr

Read the values of the Command Word, gap time, time tag, Interrupt Status Register and Transmission Status Register from the remote terminal control block where these values are posted after each transfer. These values can be used to check whether the transfer was successful, etc.

esimMil1553RtWrite

Write data to the transfer buffers of the MIL1553 device *unit* and in the **MODE_MRTMON** mode. The remote terminal *rt* and sub-address *sa* specify which write buffer must be written with the contents of *buf*. The size of the copy *len* is in words (16 bits). esimMil1553RtWrite returns 0 iff. the write is succeeded, -1 otherwise.

esimMil1553BmAdd

Append a bus monitor buffer (control block) to the MIL1553 device *unit* and in the **MODE_MRTMON** mode. The remote terminal to monitor is specified by *rt*. esimMil1553BmAdd returns 0 iff. the remote terminal is appended to the scenario (control blocks), -1 otherwise.

esimMil1553BmReadFirst

Read the first record from the ring buffer where data is posted when using the bus monitor functionality. This function must be called first before using the function **esimMil1553BmReadNext**. The *cursor* parameter is used to keep track of the current position in the ring buffer. The cursor is initialised by this function. The monitor data record format is described in the VMIC manual on page 4-166.

esimMil1553BmReadNext

Read the next record from the ring buffer where data is posted when using the bus monitor functionality. See **esimMil1553BmReadFirst** for a description of the other parameters.

esimMil1553SetMajor

Set the major time frame of MIL1553 device *unit* to *frame*. The device should be open and in BC Mode. Returns 0 if succeeded, -1 otherwise.

esimMil1553SetMinor

Set the minor time frame of MIL1553 device *unit* to *frame*. The device should be open and in BC Mode. Returns 0 if succeeded, -1 otherwise.

esimMil1553RtSetRegister

Set default register value for register *name* to the new value *value* for the remote terminal MIL1553 device *unit*. Returns 0 iff succeeded, -1 otherwise. The MIL1553 device *unit* should be open and in RT Mode.

esimMil1553SetXioRegister

Set XIO registers on the VMIC board. There are approx. 80 XIO registers. The complete list of registers are described on pages 4-9 to 4-95 in the VMIC manual. The various values for the parameter BInameBR can be found in the include file <vmic/libmil.h>. Each register has been annotated with an R and/or a W to indicate that the register is readable and/or writable.

esimMil1553GetXioRegister

Get XIO register values from the VMIC board. For information on the parameters see **esimMil1553SetXioRegister**.

esimMil1553WriteMrtmonRam

Write a value to a MRTMON RAM location. The MRTMON RAM is used in MRTMON mode. To change the behaviour of the RT during run-time the contents of the MRTMON RAM must be modified. Pages 4-170 to 4-182 of the VMIC manual describe some of the registers. The following registers which are not described in the VMIC manual can be used to generate errors from the RT side. The error generation selects are stored at locations beginning at 320. One location for each RT through location 33F. The error generation word bit maps are stored starting at 340. Two locations for each RT through location 37F. The first two locations for RT 0, and the last two for RT 31.

esimMil1553ReadMrtmonRam

Read a value from a MRTMON RAM location. See also **esimMil1553WriteMrtmonRam**.

esimMil1553Selftest

Perform the built-in selftest of the VMIC board. The result of the test is stored in the Active Status Register. Its contents are returned in the parameter *asr*.

esimMil1553BcGetCb

Returns a pointer to the bus controller control block in the VMIC static RAM area. Changing data in this memory area will change the behaviour of this control block during run-time.

esimMil1553RtGetCb

Returns a pointer to the remote terminal control block in the VMIC static RAM area. Changing data in this memory area must be done before calling **esimMil1553Start** or else the changes will not be taken into account. When starting the card in MRTMON mode, the processor on the VMIC card will copy the contents of the RT control blocks into MRTMON RAM. Changing data after that has happened will have no effect. In order to change data in MRTMON RAM, use the **esimMil1553WriteMrtmonRam** function.

esimMil1553BcCbIter

Iterates over the list of bus controller control blocks installed on the VMIC board specified by *unit*. It calls function *callback* for each control block. The callback function receives the following arguments: *unit* is the unit number of the device, *bc_cb* is the pointer to the control block on the VMIC card and *userdata* is a pointer to user supplied data which was passed as the last argument to the call to **esimMil1553BcCbIter**.

SEE ALSO

VMIVME-6000 1553 Communications Interface Board Product Manual, doc. nr. 500-006000-000-B

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Johannes Bosman, Origin/Niewegein BV, The Netherlands.
Addition of esimMil1553SetMajor, esimMil1553SetMinor and esimMil1553RtSetRegister by Peter Schrap, NLR, The Netherlands.
Addition of esimMil1553BcGetDbufHdr, esimMil1553RtGetDbufHdr, esimMil1553BmReadFirst, esimMil1553BmReadNext, esimMil1553SetXioRegister, esimMil1553GetXioRegister, esimMil1553WriteMrtmonRa esimMil1553ReadMrtmonRam, esimMil1553Selftest, esimMil1553BcGetCb, esimMil1553RtGetCb, esimMil1553RtGetCc and esimMil1553BcCbIter by Robert H. de Vries, Dutch Space BV, The Netherlands.

2.2.7 esimLink

Functions for creating and manipulating simulated satellite communication links

SYNOPSIS

For all clients (EuroSim Model code and external programs)
cc ... -lesClient -les
Link * esimLinkOpen(const char *id, const char *mode)
typedef int (*LinkDelayProc)(Link *link, int last_delay)
typedef double (*LinkTimeProc)(void)
typedef int (*LinkNotifyProc)(int size, const void *data)
int esimLinkIoctl(Link *link, LinkIoctlCommand command, ...)
void esimLinkClose(Link *link)
int esimLinkWrite(Link *link, const void *buf, int size)
int esimLinkRead(Link *link, void *buf, int size)
int esimLinkStep(void)
void esimLinkShutdown(void)
EuroSim External Clients only
int esimLinkConnect(Link *link, SIM *connect)

DESCRIPTION

The purpose of this library is to simulate the bandwidth and time-delay that characterize a long-range communication link, such as the TM/TC link between a ground station and a satellite. This is achieved by providing a central server process running within EuroSim, via which the two terminators (or clients) can communicate. The server can maintain one or more client-to-client links; links are bi-directional and can be established between any two internal clients or between an internal and an external client. For the latter, use is made of TCP/IP. No link can be established between two external clients.
The server process is essentially passive, and only operates when called. Incoming packet processing is done explicitly by polling. There are two methods to do this. One uses **esimLinkRead**() and the other uses **esimLinkStep**() and a call-back function set by calling **esimLinkIoctl**() with *LINKNOTIFYPROC* and providing a call-back which processes the packet Links can be purged from overdue packets by calling the **esimLinkStep**() function.
The function **esimLinkOpen**() initializes a link. Parameter *id* is the name of the link. Parameter *mode* defines the read-write mode. Valid values are ”r”, ”w” and ”rw”. A point-to-point link is not established until the other side has also called **esimLinkOpen**() with the same link name.
When the process is an external client, the client **must** call **esimLinkConnect**() right after the **esimLinkOpen**() call. When this is not done the actual connection with the other TM/TC client is not established and traffic will not be possible. The first argument is the link that has just been created and the second argument is the **SIM** pointer that was returned from **extConnect**(). The function **esimLinkWrite**() is used to write data onto the link. Parameter *link* is the TM/TC link handle. Parameter *buf* is a pointer to the data to write. Parameter *size* is the number of bytes to write. The data can be of any size and kind. A successful write doesn't mean the data actually reached the other side of the link as data will only be available for reading there after the links time delay has expired. By default data written to an unconnected link is lost. If this is not desired the number of packets to be saved can be specified using the **esimLinkIoctl**() with argument LINKWRITEQUEUESIZE and the number of packets as parameter. In case the data is discarded the return value is 0, otherwise it is the number of bytes written to the link.
The function **esimLinkRead**() reads packets from the link. Parameter *link* is a pointer to the TM/TC link handle. Parameter *buf* is a pointer to a buffer which should be large enough to store the package in. Parameter *size* is the size of the buffer in bytes. If there is no data to read the function will return immediately with value 0. When the buffer is large enough to hold the packet, the packet is copied into the buffer and the return value is the size of the packet read. If the buffer size is not large enough -2 will be returned. This should not happen under normal circumstances as the maximum size of a TM/TC

packet should be known in advance. It is of course possible to try to read the packet again using a larger buffer.
Another method to read packets is by using the LINKNOTIFYPROC call-back function. This function is called by the function **esimLinkStep**() every time a packet is available.
The function **esimLinkStep**() is called to drain all overdue packets of all open links and, if a link has a call-back function (see **esimLinkIoctl**()), call the call-back function for each package that is due to arrive.
The function **esimLinkIoctl**() is used to set some options for the given link, and to retrieve information about the link itself. The following commands are possible (Third argument also listed):

LINKDELAY

Argument: **integer**

Set the delay of the packets to the given number of milliseconds.

LINKDELAYPROC

Argument: **delayproc**

The given function will be called for each time a new package is put on the link with the call **esimLinkWrite**(). Its return value is used as the delay for the given package. The arguments it gets are the Link identifier for which the delay is requested, and the last delay returned by this function for this Link.

LINKSIMTIMEPROC

Argument: **timeproc**

The given function will be used throughout the module to retrieve the current simulation time. For external clients this function is default **gettimeofday**(). For internal clients, this function is default set to **rtGetSimTime**(). When an external client is linked to an internal client, the default function is changed to **gettimeofday**().

Be aware that for internal links the delay computation fails in any simulator mode other than executing and that the scheduler frequency determines the precision of the delay. When running non-realtime (i.e. slower or faster than real-time) that bandwidth restrictions and delays are not computed correctly for external links. For example if you are sending TM data with a frequency of 1 Hz to an external link you will be sending data with a rate lower or higher than 1 Hz.

LINKBANDWIDTH

Argument: **integer**

Indicates the number of bytes which can be sent over the link per second. A negative number indicates 'Unlimited' (default) and will pose NO extra delay. When this value is set to 100, and 200 bytes are sent over, the package will take 2 seconds + the LINKDELAY to arrive at the other side. Note that the package will arrive as a single entity and therefore will not be visible (and cannot be read) as long as the complete package has not arrived.

LINKSTATUS

Argument: **None**

Returns the current status of the link. This is either **LINKCONNECTED** or **LINKUNCONNECTED**. At the external client, this call will always return **LINKCONNECTED** since the status of the link is not known at the client side.

LINKAVAIL

Argument: **None**

This function returns whether a package is due to arrive now (or a short time ago). 0 (zero) means no package is available, 1 means that data is available to be read.

LINKNEXTMESSAGETIME

Argument: **None**

This function will return the time when the next message is expected to arrive. This function however is not accurate, because new data may arrive, that has a faster route, and can therefore arrive sooner (this is not a FIFO). Also packets may be incoming from the external connection that are not yet accounted for.

LINKMAXTIME_PENDING

Argument: **Integer**

Indicates how many milliseconds data may be overdue in the queue before it is discarded. When a value less than zero is given, the packets will never be discarded (default).

LINKNOTIFYHANDLER

Argument: **notifyproc**().

The given (call back) function will be called for each package that arrives and is not overdue more than **LINKMAXTIME_PENDING**. The call back function will be called for these packets, when **esimLinkStep**(). is called. This function will get the received data and its length as an argument.

LINKREADQUEUESIZE

Argument: **integer**

The maximum number of pending packets in the incoming queue of the link. (Default = -1). A value of -1 means that there is no limit to the number of packets in the queue. A warning is given when this limit is exceeded. The packet will be discarded.

LINKWRITEQUEUESIZE

Argument: **integer**

The maximum number of pending packets in the outgoing queue of the link. (Default = 0). A value of -1 means that there is no limit to the number of packets in the queue. A value of 0 means that all packets are silently discarded if the link is unconnected. A warning is given when this limit is exceeded. The packet will be discarded.

The function **esimLinkShutdown**() is used to shutdown all open connections and frees all pending data. This should only be done when the program is about to shut-down, or when the link package is not going to be used again.

EXTERNAL CLIENT LIMITATIONS

The external client is not synchronized with the EuroSim server. Because of this the network delay cannot be accounted for and some slight timing fluctuations may occur.

DIAGNOSTICS

All functions that return an int indicate errors with a negative value. A zero return indicates a successful call, except for **esimLinkWrite** where it means that the packet is lost. A successful call of **esimLinkWrite** returns the number of bytes written to the link. Functions returning a pointer will return a NULL pointer if an error occurred. Void functions can either not go wrong, or no sensible error can be returned.

SEE ALSO

extClient(3).

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Erik de Rijk, Dutch Space BV, The Netherlands.

2.2.8 esimRec: esimRecOpen, esimRecWriteRaw, esimRecClose, esimRecWriteHeader, esimRecWriteRecord, esimRecInt8FieldAdd, esimRecUint8FieldAdd, esimRecInt16FieldAdd, esimRecUint16FieldAdd, esimRecInt32FieldAdd, esimRecUint32FieldAdd, esimRecInt64FieldAdd, esimRecUint64FieldAdd, esimRecFloatFieldAdd, esimRecDoubleFieldAdd, esimRecInt8ArrayFieldAdd, esimRecUint8ArrayFieldAdd, esimRecInt16ArrayFieldAdd, esimRecUint16ArrayFieldAdd, esimRecInt32ArrayFieldAdd, esimRecUint32ArrayFieldAdd, esimRecInt64ArrayFieldAdd, esimRecUint64ArrayFieldAdd, esimRecFloatArrayFieldAdd, esimRecDoubleArrayFieldAdd

EuroSim user callable functions to create user defined recorders (C interface)

SYNOPSIS

#include <esimRec.h>
cc ... -L$EFOROOT/lib -lesServer -les
EsimRec* esimRecOpen(const char *_path_, int _flags_);
int esimRecWriteRaw(EsimRec *_rec_, const void *_ptr_, size_t _size_);
int esimRecWriteHeader(EsimRec *_rec_);
int esimRecWriteRecord(EsimRec *_rec_);
int esimRecClose(EsimRec *_rec_);

Recorder field addition routines

int esimRecInt8FieldAdd(EsimRec *_rec_, const char *_name_, int8_t *_address_);
int esimRecUint8FieldAdd(EsimRec *_rec_, const char *_name_, uint8_t *_address_);
int esimRecInt16FieldAdd(EsimRec *_rec_, const char *_name_, int16_t *_address_);
int esimRecUint16FieldAdd(EsimRec *_rec_, const char *_name_, uint16_t *_address_);
int esimRecInt32FieldAdd(EsimRec *_rec_, const char *_name_, int32_t *_address_);
int esimRecUint32FieldAdd(EsimRec *_rec_, const char *_name_, uint32_t *_address_);
int esimRecInt64FieldAdd(EsimRec *_rec_, const char *_name_, int64_t *_address_);
int esimRecUint64FieldAdd(EsimRec *_rec_, const char *_name_, uint64_t *_address_);
int esimRecFloatFieldAdd(EsimRec *_rec_, const char *_name_, float *_address_);
int esimRecDoubleFieldAdd(EsimRec *_rec_, const char *_name_, double *_address_);

Recorder array field addition routines

int esimRecInt8ArrayFieldAdd(EsimRec *_rec_, const char *_name_, size_t _n_elem_, int8_t *_address_);
int esimRecUint8ArrayFieldAdd(EsimRec *_rec_, const char *_name_, size_t _n_elem_, uint8_t *_address_);
int esimRecInt16ArrayFieldAdd(EsimRec *_rec_, const char *_name_, size_t _n_elem_, int16_t *_address_);
int esimRecUint16ArrayFieldAdd(EsimRec *_rec_, const char *_name_, size_t _n_elem_, uint16_t *_address_);
int esimRecInt32ArrayFieldAdd(EsimRec *_rec_, const char *_name_, size_t _n_elem_, int32_t *_address_);
int esimRecUint32ArrayFieldAdd(EsimRec *_rec_, const char *_name_, size_t _n_elem_, uint32_t *_address_);
int esimRecInt64ArrayFieldAdd(EsimRec *_rec_, const char *_name_, size_t _n_elem_, int64_t *_address_);
int esimRecUint64ArrayFieldAdd(EsimRec *_rec_, const char *_name_, size_t _n_elem_, uint64_t *_address_);
int esimRecFloatArrayFieldAdd(EsimRec *_rec_, const char *_name_, size_t _n_elem_, float *_address_);
int esimRecDoubleArrayFieldAdd(EsimRec *_rec_, const char *_name_, size_t _n_elem_, double *_address_);

DESCRIPTION

esimRecOpen() opens a user-defined recorder file. The file is opened for writing. If the *path* is relative, the file is created in the recorder directory. The *flags* parameter contains configuration and/or option flags. It shall be set to 0 if no options are selected. The recorder handle is returned. On error NULL is returned.

After calling **esimRecOpen**() the user can choose to write a EuroSim native recorder file or to write a completely user-defined file. The difference is made by the calls following the **esimRecOpen**() call. If you start using **esimRecWriteRaw**() then you write a completely user-defined file. If you start using **esimRec*FieldAdd**() and **esimRec*ArrayFieldAdd**() calls followed by **esimRecWriteHeader**() and **esimRecWriteRecord**() then you are writing a EuroSim native recorder file. You are not allowed to mix the two styles of recorder files. An error return value will be issued if you try to do so.
esimRecWriteRaw() writes the *size* bytes of data in *ptr* to the recorder file indicated by the recorder handle *rec*.
esimRecWriteHeader() writes the recorder file header to disk. The simulation time is automatically included as the first field of each recording. After calling this function no more fields can be added to the recorder. Only calls to **esimRecWriteRecord**() and **esimRecClose**() are allowed. *rec* is the recorder file handle.
esimRecWriteRecord() samples all the variables that are in the recording referenced by *rec* and writes it to disk.
esimRec*type***FieldAdd**(), where *type* can be Int8, Uint8, Int16, Uint16, Int32, Uint32, Int64, Uint64, Float or Double, is used to add a data field to the recorder of the specified type. *rec* is the recorder file handle. *name* is the symbolic name of the field. *address* is the address pointing to the variable to be recorded.
esimRec*type***ArrayFieldAdd**(), where *type* can be Int8, Uint8, Int16, Uint16, Int32, Uint32, Int64, Uint64, Float or Double, is used to add an array data field to the recorder of the specified type. *rec* is the recorder file handle. *name* is the symbolic name of the field. *n_elem* is the number of elements in the array. *address* is the address pointing to the variable to be recorded.
esimRecClose() closes the user-defined recorder file indicated by recorder handle *rec*.

RETURN VALUE

In case the return type is an integer, 0 means success, -1 means a general failure, -2 means that calls to esimRecWriteRaw are used to write to a EuroSim native recorder file or that the order of calls to create a native EuroSim recorder file header is incorrect, -3 means that one or more calls have failed at the asynchronous side in the past. It is not possible to return failure codes synchronously as that would require the call to perform non-real-time tasks (file I/O) in the real-time domain. Therefore the next best thing is done namely to report the failure when performing another call on the synchronous side.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert de Vries, Dutch Space BV, The Netherlands.

2.2.9 esimSerialOpen, esimSerialClose, esimSerialRead, esimSerialWrite, esimSerialFlush, esimSerialInstall

General operations for serial Unix devices.

SYNOPSIS

#include <esimSerial.h>
int esimSerialOpen(unsigned char *name, int buffered);
bool esimSerialClose(int fd);
int esimSerialRead(int fd, unsigned char *buf, int nbytes);
bool esimSerialWrite(int fd, unsigned char *buf, int nbytes);
void esimSerialFlush(int fd);
bool esimSerialInstall(int fd, esimSerial_Handler handler);

DESCRIPTION

The unit *esimSerial* provides a general interface to access the standard RS232/RS422 serial devices. Note that parallel access to the same device from different EuroSim tasks is not allowed, this because the functions are non blocking. A simultaneous read/write to the same device has to be solved in the EuroSim Schedule.

ACCESS ROUTINES

esimSerialOpen

Opens a serial device with the name specified by *name*. The flag *buffered* indicates whether the device is required to buffer its input or not. The default serial interface settings are: 19200 baud, 8 databits, 1 stopbit, no parity, hang up on last close, local line, line discipline raw, non-blocking. All pending input and output data is flushed. Returns the file descriptor iff. the device is opened, -1 otherwise.

esimSerialClose

Closes the serial device specified by *fd*. Returns TRUE iff. the serial device is closed, FALSE otherwise.

esimSerialRead

Read *nbytes* bytes from serial device *fd* into *buf*. Returns the number of bytes read, returns 0 if no bytes are read, or a negative number when an error occurred.

esimSerialWrite

Write *nbytes* bytes to serial device *fd* from *buf*. Returns TRUE iff. the bytes are written, FALSE otherwise.

esimSerialFlush

Flushes the serial device *fd*.

esimSerialInstall

Install a signal handler *handler* for the *SIGPOLL* signal for the serial device *fd*. Returns TRUE iff. the handler is installed, FALSE otherwise.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Johannes Bosman, Origin/Niewegein BV, The Netherlands.

2.2.10 esim: esimgetsimtime, esimsetsimtime, esimgetsimtimeutc, esimsetsimtimeutc, esimgetsimtimeymdhmss, esimsetsimtimeymdhmss, esimgetwallclocktime, esimgethighreswallclocktime, esimgetstate, esimsetstate, esimsetstatetimed, esimgetmaincycletime, esimgetmaincycleboundarysimtime, esimgetmaincycleboundarywallclocktime, esimgettaskname, esimgettaskrate, esimsetclockfrequency, esimgetclockfrequency, esimsetspeed, esimgetspeed, esimeventraise, esimeventraisetimed, esimeventcount, esimeventdata, esimdisabletask, esimenabletask, esimentrypointfrequency, esimgetrealtime, esimsetrealtime, esimgetrecordingstate, esimsetrecordingstate esimversion esimreport, esimmessage, esimwarning, esimerror, esimfatal, esimabortnow

EuroSim user callable functions (Fortran interface)

SYNOPSIS

include 'esim.inc'
f77 ... -L$EFOROOT/lib -lesServer -les

REALTIME TIMING FUNCTIONS

double precision time, timeYMDHMS(7)
integer ok, timespec(2), timeYMDHMS(7)
time = esimgetsimtime
call esimgetsimtimets(timespec)
call esimgetsimtimeymdhmss(timeYMDHMS)
ok = esimsetsimtime
ok = esimsetsimtimets(timespec)
ok = esimsetsimtimeymhdmss(timeYMDHMS)
time = esimgetwallclocktime
time = esimgetwallclocktimets(timespec)
time = esimgethighreswallclocktime

REALTIME SIMULATION STATE FUNCTIONS

integer state, ok, t(2), use_simtime
state = esimgetstate
ok = esimsetstate(state)
ok = esimsetstatetimed(state, t, use_simtime)
call esimgetmaincycletime(t)
call esimgetmaincycleboundarysimtime(t)
call esimgetmaincycleboundarywallclocktime(t)

REALTIME TASK RELATED FUNCTIONS

double precision rate, freq
integer ok, count, on, size, data(N), state
character*N eventname, taskname, entrypoint
call esimgettaskname(taskname)
rate = esimgettaskrate
ok = esimenabletask(taskname)
ok = esimdisabletask(taskname)
ok = esimentrypointfrequency(state, entrypoint, freq)
on = esimgetrealtime
ok = esimsetrealtime(on)

ok = esimeventraise(eventname, data, size)
ok = esimeventraisetimed(eventname, data, size, t, use_simtime)
ok = esimeventdata(data, size)
count = esimeventcount(eventname)

REALTIME CLOCK FUNCTIONS

double precision speed
integer ok
speed = esimgetspeed
ok = esimsetspeed(speed)

REALTIME RECORDING FUNCTIONS

integer on, ok
on = esimgetrecordingstate
ok = esimsetrecordingstate(on)

REALTIME REPORTING FUNCTIONS

character*N message
integer level
call esimmessage(message)
call esimwarning(message)
call esimerror(message)
call esimfatal(message)
call esimreport(level, message)

AUXILIARY FUNCTIONS

character*N version
call esimversion(version)
call esimabortnow()

DESCRIPTION

Required interface for simulation models that want to use the EuroSim services. These calls can be used both from three programming languages: C, Fortran and Ada (see **esim**(3C), **esim_ftn**(3) and **esim_ada**(3)).
esimgetsimtime returns the simulation time in seconds with the precision of the basic period with which the simulation runs (5 ms by default). In case the simulation is driven by the external interrupt the precision is equal to that period. If the simulator has real-time errors the simulation time will be slower than the wall clock. The simulation time is set to zero (0) on arriving in initialising state. **esimgetsimtimeutc** returns the simulation time in a timespec structure. **esimgetsimtimeymdhmss** returns the simulation time in an array of 7 int's containing: year, month, day, hour, minute, second and nanoseconds.
esimsetsimtime sets the requested simulation time *simtime* in seconds. This can only be done in the standby state. If calling **esimSetSimtime** in any other state is attempted or *simtime* is less than zero, no simulation time is set and (-1) is returned. On success zero (0) is returned. **esimsetsimtimeutc** sets the simulation time using a timespec structure. **esimsetsimtimeymdhmss** set the simulation time using an array of 7 int's containing: year, month, day, hour, minute, second and nanoseconds.
esimgetwallclocktime returns the wall clock time in seconds. The basic resolution is equal to the resolution of the high-res time described next, but it is truncated to milliseconds. The wall clock time is set to zero (0) when the first model task is scheduled, and runs real-time which means that is independent from the simulation time.

esimgethighreswallclocktime returns the "same" time as **esimgetwallclocktime** but in milliseconds and with a higher resolution. This high resolution is 21 ns on high-end platforms equipped with a hardware timer such as a challenge and onyx. On low end platforms this resolution as can be achieved by the **gettimeofday**(3B) call. For more information on resolutions of this call see **timers**(5). **esimgetwallclocktimeutc** returns the wall clock time in a timespec structure.
esimgetstate returns the current simulator state. The state can be any of the following values: *esimUnconfiguredState*, *esimInitialisingState*, *esimExecutingState*, *esimStandbyState* or *esimStoppingState*.
esimsetstate sets the simulator state to the indicated value *state*. *state* can be any of the following values: *esimUnconfiguredState*, *esimInitialisingState*, *esimExecutingState*, *esimStandbyState* or *esimStoppingState*. If *state* is not reachable from the current state *0* is returned; on a successful state transition *1* is returned.
esimsetstatetimed sets the simulator state to the indicated value *state* at the specified time *t*. The possible values of *state* are listed in the previous paragraph. If the flag *use_simtime* is set to 1 (true), the specified time is interpreted as simulation time. If the flag is set to 0 (false), the specified time is interpreted as the wallclock time. The transition time uses a struct timespec where the number of seconds is relative to January 1, 1970. On success this function returns 0, otherwise -1.
esimgetmaincycletime returns the main cycle time of the schedule. The result can be used to compute valid state transition times for use in the function esimsetstatetimed.
esimgetmaincycleboundarysimtime returns the simulation time of the last state transition. This boundary time can be used to compute valid state transition times for use in the function esimsetstatetimed when the value of use_simtime is true.
esimgetmaincycleboundarywallclocktime returns the wallclock time of the last state transition. This boundary time can be used to compute valid state transition times for use in the function esimsetstatetimed when the value of use_simtime is false.
esimgettaskname returns the name of your current task.
esimgettaskrate returns the frequency (in Hz) of your current task.
esimentrypointfrequency stores the frequency (in Hz) of the entrypoint with the name *entrypoint* in the argument *freq* in the state *state*. If the entrypoint appears multiple times in the schedule the function returns -1. If the entrypoint does not appear in the schedule in the given state, the frequency is 0.
esimdisabletask disables the task *taskname* defined with the Schedule Editor **ScheduleEditor**(1). It will be skipped (not executed) by the EuroSim runtime until a call is made to **esimenabletask**.
esimenabletask enables the task *taskname* defined with the Schedule Editor **ScheduleEditor**(1). It will be executed/scheduled according to the schedule made with the Schedule Editor.
esimgetrealtime returns the current operational state of the EuroSim realtime Scheduler. If *1* is returned, hard realtime execution is in progress, whereas a return of *0* indicates that your model is not executing in realtime mode.
esimsetrealtime sets the current operational state of the EuroSim realtime Scheduler. Hard real time execution can only be set if the scheduler was launched in hard real time mode. *1* is returned on success. *0* is returned on failure.
esimeventraise raises the event *eventname* for triggering tasks defined with the Schedule Editor **ScheduleEditor**(1). User defined data can be passed in *data* and *size*. On success this function returns 0, otherwise -1.
esimeventraisetimed raises the event *eventname* for triggering tasks defined with the Schedule Editor **ScheduleEditor**(1) at the specified time *t*. User defined data can be passed in *data* and *size*. If the flag *use_simtime* is set to 1 (true), the specified time is interpreted as simulation time. If the flag is set to 0 (false), the specified time is interpreted as the wallclock time. The transition time uses a struct timespec where the number of seconds is relative to January 1, 1970. On success this function returns 0, otherwise -1.
esimeventcount returns the number of times that event *eventname* has been raised, or -1 if no such event is defined.
esimeventdata returns the event data that was passed with the event that activated the input connector connected to the current task. The input value of *size* specifies the size of the *data* buffer. The output value of *size* is the size of the retrieved data. On success this function returns 0, otherwise -1.

esimgetspeed returns the current speed of EuroSim Scheduler. e.g. *1.0* means (hard or soft) real time. *0.1* means slowdown by a factor 10. *-1* means as fast as possible.
esimsetspeed sets the current speed of EuroSim Scheduler. e.g. *1.0* means (hard or soft) real time. *0.1* means slowdown by a factor 10. *-1* means as fast as possible. The speed can only be changed if the scheduler is running non realtime. If *speed* is not a feasible speed *0* is returned; on a successful setting of the speed *1* is returned.
esimgetclockfrequency has been replaced by *esimgetspeed*.
esimsetclockfrequency has been replaced by *esimsetspeed*.
esimgetrecordingstate returns the current state of the EuroSim realtime data Recorder. If *1* is returned, data is logged to disk, whereas a return of *0* indicates that recording is switched off.
esimsetrecordingstate sets the state of the Recorder to *on*. If *on* is *1* data will subsequently be written to disk, if *on* is *0* data recording will be suspended. Return value is either *1* or *0*, depending on success or not.
The functions **esimreport**, **esimmessage**, **esimwarning**, **esimerror** and **esimfatal** can be used to send messages from the EuroSim Model code to the test-conductor interface. The **esimreport** function allows the caller to specify the severity of the message. The other functions have implicit severities. The possible severity levels are:

```
esimSevMessage  comment or verbose information
esimSevWarning  warning messages
esimSevError    error messages
esimSevFatal    non-recoverable errors
```

esimversion places a string in parameter version indicating the current version of EuroSim that you are running.
esimabortnow immediately starts termination and cleanup of the simulator. This is useful when an error condition is found (f.i. at initialisation time) and no more entrypoints should be scheduled for execution.

NOTES

libesServer.so contains all EuroSim realtime libraries including those from *ActionMgr* (see **actionmgr**(3)), *Scheduler* (see **Sched_***(3)), *Recorder* and *Stimulator* (see **recstim**(3)), *Script* (see **script**(3)), etcetera.....

SEE ALSO

esim(3C)

WARNINGS

The message handling routines use ring buffers to decouple the realtime and non realtime domains of EuroSim. The size of these ring buffers and therefore the maximum message size is limited (currently 2*BUFSIZ as defined in **stdio.h**). Messages may be lost if the message rate is too high for the message handler.

BUGS

We'll need a **esimsetwallclocktime**, **esimgettaskoffset** and *esimsettaskoffset* one day.
A non-real-time non-parallel version of this library should be made.
Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Wim Elhorst, Marc Neefs and Robert de Vries for Dutch Space BV, The Netherlands.

2.2.11 esim: EsimMalloc, EsimFree, EsimRealloc, EsimCalloc, EsimStrdup, EsimGetSimtime, EsimSetSimtime, EsimGetSimtimeUTC, EsimSetSimtimeUTC, EsimGetSimtimeYMDHMSs, EsimSetSimtimeYMDHMSs, EsimGetHighResWallclocktime, EsimGetWallclocktime, EsimGetState, EsimSetState, EsimSetStateTimed, EsimGetMainCycleTime, EsimGetMainCycleBoundarySimtime, EsimGetMainCycleBoundaryWallclocktime, EsimGetTaskname, EsimGetTaskrate, EsimSetClockfrequency, EsimGetClockfrequency, EsimSetSpeed, EsimGetSpeed, EsimEventRaise, EsimEventRaiseTimed, EsimEventCount, EsimEventData EsimDisableTask, EsimEnableTask, EsimGetRealtime, EsimSetRealtime, EsimGetRecordingState, EsimSetRecordingState, EsimVersion, EsimReport, EsimMessage, EsimWarning, EsimError, EsimFatal

EuroSim user callable functions (Ada interface)

SYNOPSIS

use Esim; with Esim;

REALTIME (SHARED) MEMORY ALLOCATION

function EsimMalloc(Size : Size_T) return Void_Ptr
procedure EsimFree(Ptr : Void_Ptr)
function EsimRealloc(Ptr : Void_Ptr Size : Size_T) return Void_Ptr
function EsimCalloc(Nelem : Size_T Elsize : Size_T) return Void_Ptr
function EsimStrdup(Str : Chars_Ptr) return Chars_Ptr
function EsimStrdup(Str : String) return String

REALTIME TIMING FUNCTIONS

function EsimGetSimtime return Long_Float
function EsimSetSimtime(Simtime: Long_Float) return Integer
function EsimGetSimtimeUTC return Time_Spec;
function EsimSetSimtimeUTC(Simtime: in Time_Spec) return Integer;
procedure EsimGetSimtimeYMDHMSs(SimTime: out YMDHMSs);
function EsimSetSimtimeYMDHMSs(Simtime: in YMDHMSs) return Integer;
function EsimGetWallclocktime return Long_Float
function EsimGetHighResWallclocktime return Long_Float
function EsimGetWallclocktimeUTC return Time_Spec;

REALTIME SIMULATION STATE FUNCTIONS

function EsimGetState return esimState
function EsimSetState(State: esimState) return Integer
function EsimSetState(State : esimState) return Boolean
function EsimSetStateTimed(State: EsimState; T: in Time_Spec; Use_Simtime: Integer) return Integer;
function EsimSetStateTimed(State: EsimState; T: in Time_Spec; Use_Simtime: Boolean) return Boolean;
function EsimGetMainCycleTime return Time_Spec;
function EsimGetMainCycleBoundarySimtime return Time_Spec;
function EsimGetMainCycleBoundaryWallclocktime return Time_Spec;

REALTIME TASK RELATED FUNCTIONS

function EsimGetTaskname return Chars_Ptr
function EsimGetTaskname return String
function EsimGetTaskrate return Long_Float;
function EsimEnableTask(TaskName : Chars_Ptr) return Integer
function EsimEnableTask(TaskName : String) return Boolean
function EsimDisableTask(TaskName : Chars_Ptr) return Integer
function EsimDisableTask(TaskName : String) return Boolean
function EsimGetRealtime return Boolean
function EsimSetRealtime(On : Integer) return Integer;
function EsimSetRealtime(On : Boolean) return Boolean;

EVENT FUNCTIONS

function EsimRaiseEvent(EventName : in Chars_Ptr) return Integer
function EsimRaiseEvent(EventName : in String) return Boolean
function EsimEventRaise(EventName : in Chars_Ptr; Data : in Void_Ptr; Size : Integer) return Integer;
function EsimEventRaise(EventName: in String; Data : in Void_Ptr; Size : Integer) return Boolean;
function EsimEventRaiseTimed(EventName : in Chars_Ptr; Data : in Void_Ptr; Size : Integer; T : in Time_Spec; Use_Simtime : Integer) return Integer;
function EsimEventRaiseTimed(EventName : in String; Data : in Void_Ptr; Size : Integer; T : in Time_Spec; Use_Simtime : Boolean) return Boolean;
type Integer_Ptr is access Integer;
function EsimEventData(Data : in Void_Ptr; Size : Integer_Ptr) return Integer;
function EsimEventCount(EventName : Chars_Ptr) return Integer;
function EsimEventCount(EventName : String) return Integer

REALTIME CLOCK FUNCTIONS

function EsimGetSpeed return Long_Float;
function EsimSetSpeed(Frequency : Long_Float) return Integer;
function EsimSetSpeed(Frequency : Long_Float) return Boolean;
function EsimGetClockfrequency return Long_Float
function EsimSetClockfrequency(Frequency : Long_Float) return Integer
function EsimSetClockfrequency(Frequency : Long_Float) return Boolean

REALTIME RECORDING FUNCTIONS

function EsimGetRecordingState return Integer
function EsimGetRecordingState return Boolean
function EsimSetRecordingState(On : Integer) return Integer
function EsimSetRecordingState(On : Boolean) return Boolean

REALTIME REPORTING FUNCTIONS

procedure esimMessage(Message : Chars_Ptr)
procedure esimMessage(Message : String)
procedure esimWarning(Warning : Chars_Ptr)
procedure esimWarning(Warning : String)
procedure esimError(Error : Chars_Ptr)
procedure esimError(Error : String)
procedure esimFatal(Fatal : Chars_Ptr)
procedure esimFatal(Fatal : String)
procedure esimReport(S : esimSeverity Report : Chars_Ptr)

procedure esimReport(S: esimSeverity Report : String)

AUXILIARY FUNCTIONS

function EsimVersion return Chars_Ptr
function EsimVersion return String

DESCRIPTION

Required interface for simulation models that want to use the EuroSim services from Ada. This manual page summarizes the synopsis for the Ada syntax. Consult the Ada specification file *$EFOROOT/adainclude/esim.ads* for syntactical details.
An in-depth description of the calls can be found in the **esim**(3C) manual page, where the C versions of the synonymous functions are described.

SEE ALSO

esim(3C) and **esim**(3F)

FILES

$EFOROOT/adainclude/esim.ads

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert de Vries for Dutch Space BV, The Netherlands.

2.2.12 esimJava

Example of the EuroSim publication interface for Java.

SYNOPSIS

import esimJava.*;
javac -classpath $EFOROOT/lib/java/esimJava.jar ...
class Publication

DESCRIPTION

This gives an example of the EuroSim publication interface for Java models. In the first class a model called ”MyModel” is defined. This model contains a variable with a eurosim annotation and an entry-point with a eurosim annotation. The publication process uses reflection to determine the variables and entrypoints of the model.
The second class called ”main” is required to instantiate models and it serves as the entrypoint for the publication mechanism.

EXAMPLE

```
public class MyModel {
    @eurosim(description="variable description")
    public int a;

    public MyModel() {
        a = 1;
    }

    @eurosim(description="entrypoint description")
    public void hello()
    {
        a++;
        String msg = "a = " + a;
        EsimRuntime.esimMessage(msg);
    }
}

public class main {
    @eurosim(description="Model Instance 1")
    MyModel m1 = new MyModel();

    @eurosim(description="Model Instance 2")
    MyModel m2 = new MyModel();
}
```

SEE ALSO

javadictgen(2), **Publication**(3), **EsimRuntime**(3)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Conrad Tinkler, Atos Origin Technical Automation, The Netherlands

2.2.13 EsimRuntime: esimMessage, esimWarning, esimError, esimFatal, esimGetSimTime, esimGetWallclocktime, esimGetState, esimSetState, esimEventRaise, esimDisableTask, esimEnableTask,

EuroSim user callable methods (Java interface)

SYNOPSIS

import esimJava.*;
javac -classpath $EFOROOT/lib/java/esimJava.jar ...
class EsimRuntime

DESCRIPTION

Library class providing methods which can be called by java simulation models at runtime.
Methods:

void esimMessage(String msg);

Send a string value to be displayed in the output window of the Simulation controller.

msg = message string

void esimWarning(String msg);

Send a string value to be displayed in the output window of the Simulation controller.

msg = message string

void esimError(String msg);

Send a string value to be displayed in the output window of the Simulation controller.

msg = message string

void esimFatal(String msg);

Send a string value to be displayed in the output window of the Simulation controller.

msg = message string

int esimSetSimtime(double simtime);

Sets the requested simulation time *simtime* in seconds. This can only be done in the standbye state.

returns: 0 if successful, -1 when unsuccessful

double esimGetSimtime();

Get current simulation time

returns: a double containing the simulation time in seconds

double esimGetWallclocktime();

Get current wallclocktime

returns: a double containing the wallclocktime in seconds

esimState esimGetState();

Get the current simulation state.

returns: the simulation state.

boolean esimSetState(esimState state);

Set the simulation state.

returns: true if successful, false if not.

int esimEventRaise(String eventName, byte[] data);

Raise event with the name *eventName* in the schedule.User defined data can be passed in *data*.

returns: -1 if it fails, 0 if it succeeds.

int esimEventData(byte[] data);

Returns the event data that was passed with the event that activated the input connector connected to the current task.

returns: 0 if successful, -1 if not.

int esimDisableTask(String taskName);

Disable task *taskName* in the schedule.

returns: -1 if it fails, 0 if it succeeds.

int esimEnableTask(String taskName);

Enable task *taskName* in the schedule.

returns: -1 if it fails, 0 if it succeeds.

double esimGetTaskrate();

returns the frequency (in Hz) of your current task.

returns: task frequency

String esimGetTaskname();

returns the name of your current task.

returns: A string with the name of the current task

int esimGetRealtime();

returns the current operational state of the EuroSim realtime Scheduler.

returns: 1 if hard realtime execution is in progress, 0 when not executing in realtime mode

int esimSetRealtime(int on);

Sets the current operational state of the EuroSim realtime Scheduler. Hard realtime execution can only be set if the scheduler was launched in hard real time mode.

returns: 1 if success, 0 is returned on failure.

int esimSetSpeed(double speed);

Sets the current speed of EuroSim Scheduler. e.g. *1.0* means (hard soft) realtime. *0.1* means slowdown by a factor 10. *-1* means as fast as possible. The speed can only be changed if the scheduler is running non realtime.

returns: 0 if *speed* is not a feasable speed, 1 if set successfully

double esimGetSpeed();

Returns the current speed of EuroSim Scheduler.

returns: The current speed

int esimGetRecordingState();

Returns he current state of the EuroSim realtime data Recorder.

returns: 1 if data is logged to disk, 0 when not recording

int esimSetRecordingState(int on);

Sets the state of the Recorder to *on*, if *on* is 1 data will subsequently be written to disk, if *on* is 0 data recording will be suspended.

returns: 1 if success, 0 if unsuccessful

void esimAbortNow();

Immediately starts termination and cleanup of the simulator. This is useful when an error condition is found (e.g. at initialisation time) and no more entrypoints should be scheduled for execution.

String esimVersion();

Returns a string indicating the current version of EuroSim that you are running.

returns: The currently running EuroSim version

SEE ALSO

esimJava(3), **javadictgen**(2)

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Conrad Tinkler, Atos Origin Technical Automation, The Netherlands

2.2.14 list_sessions_1, start_session_1, kill_session_by_pid_1, kill_session_by_name_1, proc_allocate_1, rename_proc_by_pid_1, check_license_1, list_sessions_2, start_sessions_2, kill_session_by_pid_2, kill_session_by_name_2, proc_allocate_2, rename_proc_by_pid_2, check_license_2, query_host_2, list_sessions_3, start_sessio kill_session_by_pid_3, kill_session_by_name_3, proc_allocate_3, rename_proc_by_pi check_license_3, query_host_3, list_sessions_4, start_session_4, kill_session_by_pi kill_session_by_name_4, proc_allocate_4, rename_proc_by_pid_4, check_license_4, query_host_4

EuroSim daemon RPC client interface functions and types

C SYNOPSIS

#include <esimd.h>
cc ... -les

Protocol version 1 (Mk1/Mk2)

session_array* list_sessions_1(void *_not_used_, CLIENT *_clnt_);
struct start_result* start_session_1(session_def *_session_, CLIENT *_clnt_);
int* kill_session_by_pid_1(kill_pid *_pgid_, CLIENT *_clnt_);
int* kill_session_by_name_1(kill_name *_name_, CLIENT *_clnt_);
int* proc_allocate_1(sim_pid *_pgid_, CLIENT *_clnt_);
int* rename_proc_by_pid_1(rename_pid *_name_, CLIENT *_clnt_);
int* check_license_1(license_query *_id_, CLIENT *_clnt_);

Protocol version 2 (Mk3/Mk3rev1)

session2_array* list_sessions_2(void *_not_used_, CLIENT *_clnt_);
struct start_result* start_session_2(struct session2_def *_session_, CLIENT *_clnt_);
int* kill_session_by_pid_2(struct kill_pid *_pgid_, CLIENT *_clnt_);
int* kill_session_by_name_2(struct kill_name *_name_, CLIENT *_clnt_);
int* proc_allocate_2(sim_pid *_pgid_, CLIENT *_clnt_);
int* rename_proc_by_pid_2(struct rename_pid *_name_, CLIENT *_clnt_);
struct check_result* check_license_2(struct license_query *_id_, CLIENT *_clnt_);
struct host_info* query_host_2(void *_not_used_, CLIENT *_clnt_);

Protocol version 3 (Mk3rev2)

session3_array* list_sessions_3(void *_not_used_, CLIENT *_clnt_);
struct start_result* start_session_3(struct session3_def *_session_, CLIENT *_clnt_);
int* kill_session_by_pid_3(struct kill_pid *_pgid_, CLIENT *_clnt_);
int* kill_session_by_name_3(struct kill_name *_name_, CLIENT *_clnt_);
int* proc_allocate_3(sim_pid *_pgid_, CLIENT *_clnt_);
int* rename_proc_by_pid_3(struct rename_pid *_name_, CLIENT *_clnt_);
struct check_result* check_license_3(struct license_query *_id_, CLIENT *_clnt_);
struct host_info* query_host_3(void *_not_used_, CLIENT *_clnt_);

Protocol version 4 (Mk4/Mk4rev1)

session4_array* list_sessions_4(void *_not_used_, CLIENT *_clnt_);
struct start_result* start_session_4(struct session4_def *_session_, CLIENT *_clnt_);
int* kill_session_by_pid_4(struct kill_pid *_pgid_, CLIENT *_clnt_);
int* kill_session_by_name_4(struct kill_name *_name_, CLIENT *_clnt_);
int* proc_allocate_4(sim_pid *_pgid_, CLIENT *_clnt_);
int* rename_proc_by_pid_4(struct rename_pid *_name_, CLIENT *_clnt_);

struct check_result* check_license_4(struct license_query *_id_, CLIENT *_clnt_);
struct host_info* query_host_4(void *_not_used_, CLIENT *_clnt_);

Protocol version 5 (Mk4rev1pl2/Mk4rev2)

session4_array* list_sessions_5(void *_not_used_, CLIENT *_clnt_);
struct start_result* start_session_5(struct session5_def *_session_, CLIENT *_clnt_);
int* kill_session_by_pid_5(struct kill_pid *_pgid_, CLIENT *_clnt_);
int* kill_session_by_name_5(struct kill_name *_name_, CLIENT *_clnt_);
int* proc_allocate_5(sim_pid *_pgid_, CLIENT *_clnt_);
int* rename_proc_by_pid_5(struct rename_pid *_name_, CLIENT *_clnt_);
struct check_result* check_license_5(struct license_query *_id_, CLIENT *_clnt_);
struct host_info* query_host_5(void *_not_used_, CLIENT *_clnt_);

DESCRIPTION

list_sessions_1 returns an array with descriptions of all running simulators started by the daemon. The array contains almost all information used to start the simulator. For a description of all the fields see the description for **start_session_1**.

```
struct session_info {
        char *work_dir;
        char *simulator;
        char *schedule;
        char *mission;
        char *dict;
        char *model;
        char *recorderdir;
        char *stimulatordir;
        char *initcond;
        char *exports;
        int prefcon;
        int realtime;
        char *timestamp;
        int uid;
        int gid;
        int pid;
};

typedef struct {
        u_int session_array_len;
        session_info *session_array_val;
} session_array;
```

list_sessions_2 returns an array with descriptions of all running simulators started by the daemon. The array contains almost all information used to start the simulator. For a description of all the fields see the description for **start_session_2**.

```
struct file_def {
        char *path;
        char *vers;
};
typedef struct file_def file_def;
```



```
struct session2_info {
        file_def sim;
        char *work_dir;
        char *simulator;
        file_def schedule;
        struct {
                u_int scenarios_len;
                file_def *scenarios_val;
        } scenarios;
        char *dict;
        file_def model;
        char *recorderdir;
        struct {
                u_int initconds_len;
                file_def *initconds_val;
        } initconds;
        char *exports;
        int prefcon;
        int realtime;
        char *timestamp;
        int uid;
        int gid;
        int pid;
};

typedef struct {
        u_int session2_array_len;
        session2_info *session2_array_val;
} session2_array;
```

list_sessions_3 returns an array with descriptions of all running simulators started by the daemon. The array contains almost all information used to start the simulator. For a description of all the fields see the description for **start_session_3**.

```
struct session3_info {
        file_def sim;
        char *work_dir;
        char *simulator;
        file_def schedule;
        struct {
                u_int scenarios_len;
                file_def *scenarios_val;
        } scenarios;
        char *dict;
        file_def model;
        char *recorderdir;
        struct {
                u_int initconds_len;
                file_def *initconds_val;
        } initconds;
        char *exports;
        char *timestamp;
        int prefcon;
        int uid;
```

```
        int gid;
        int pid;
        int flags;
};

typedef struct {
        u_int session3_array_len;
        session3_info *session3_array_val;
} session3_array;
```

list_sessions_4 returns an array with descriptions of all running simulators started by the daemon. The array contains almost all information used to start the simulator. For a description of all the fields see the description for **start_session_4**.

```
struct session4_info {
        file_def sim;
        char *work_dir;
        char *simulator;
        file_def schedule;
        struct {
                u_int scenarios_len;
                file_def *scenarios_val;
        } scenarios;
        char *dict;
        file_def model;
        char *recorderdir;
        struct {
                u_int initconds_len;
                file_def *initconds_val;
        } initconds;
        struct {
                u_int calibrations_len;
                file_def *calibrations_val;
        } calibrations;
        char *exports;
        char *timestamp;
        int prefcon;
        int uid;
        int gid;
        int pid;
        int flags;
};

typedef struct {
        u_int session4_array_len;
        session4_info *session4_array_val;
} session4_array;
```

list_sessions_5 returns an array with descriptions of all running simulators started by the daemon. The array contains almost all information used to start the simulator. For a description of all the fields see the description for **start_session_5**.

```
struct session5_info {
        file_def sim;
```



```
        char *work_dir;
        char *simulator;
        file_def schedule;
        struct {
                u_int scenarios_len;
                file_def *scenarios_val;
        } scenarios;
        char *dict;
        file_def model;
        char *recorderdir;
        struct {
                u_int initconds_len;
                file_def *initconds_val;
        } initconds;
        struct {
                u_int calibrations_len;
                file_def *calibrations_val;
        } calibrations;
        char *exports;
        char *alias;
        char *timestamp;
        int prefcon;
        int uid;
        int gid;
        int pid;
        int flags;
};

typedef struct {
        u_int session5_array_len;
        session5_info *session5_array_val;
} session5_array;
```

start_session_1 starts a new EuroSim Mk1/Mk2 simulator. Argument *session* is a structure containing all the information needed to launch a simulator.
Field *work_dir* specifies the current working directory of the test controller. The directory should exist and be accessible by the EuroSim daemon. Normally this is done by making the directory available through NFS in case the RPC call is performed from a different host.
Field *simulator* is the file name of the simulator executable (.exe). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
schedule is the file name of the simulator schedule file (.sched). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
mission is the file name of the mission definition file (.mdl). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
dict is the file name of the data dictionary file (.dict). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
model is the file name of the model file (.model). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*. This file is not actually used by the simulator for reading. It used for tracing purposes as a reference.
recorderdir is the path name of the directory where all recordings are stored. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
stimulatordir is the path name of the directory where all stimuli files are read from. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.

initcond is the file name of the initial condition file (.init). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
exports is the file name of the exports file (.export). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
environment is an array of environment variables in the usual format VAR=value. Normally it is sufficient to copy the entire current environment into this array. If you want to start the simulator with a custom environment setting you have to set at least the following environment variables used by EuroSim in addition to the ones used by the simulator model software.
EFOROOT should be set to the EuroSim installation directory.
EFO_SHAREDMEMSIZE is the amount of memory reserved for dynamic memory allocation. Default is 4194304 (4 MB).
PWD is the current working directory and is set to *work_dir* by the daemon if it is not present.
LD_LIBRARYN32_PATH should be set to the path of the shared libraries of EuroSim for Irix 6.5. The value is normally $EFOROOT/lib32.
LD_LIBRARY_PATH should be set to the path of the shared libraries of EuroSim for other systems than Irix 6.5 (e.g. Linux). The value is normally $EFOROOT/lib.
prefcon is set to -1 under normal circumstances, a connection number is selected by the daemon and returned on successful start-up of the simulator. Put a positive value here if you want to force the new simulator to have a specific connection number.
realtime is set to 1 for real-time simulation runs, or 0 for non-real-time runs.
umask is the umask used for creation of new files. See **umask**(2).
The UNIX credentials passed as part of the request to start the simulator are used to deduce the uid and gid under which it will run. Make sure to create a valid authentication by calling *authunix_create_default*() (see **rpc**(3)).
The result is returned as a union with a selector variable *status*. If *status* equals **ST_SUCCESS**, the connection number can be retrieved from field *prefcon*. If *status* equals **ST_ERROR**, an list of error messages can be retrieved from field *errors*.

```
struct session_def {
        char *work_dir;
        char *simulator;
        char *schedule;
        char *mission;
        char *dict;
        char *model;
        char *recorderdir;
        char *stimulatordir;
        char *initcond;
        char *exports;
        struct {
                u_int environment_len;
                env_item *environment_val;
        } environment;
        int prefcon;
        int realtime;
        int umask;
};

struct start_result {
        int status;
        union {
                int prefcon;
                error_array errors;
        } start_result_u;
```



```
};
```

start_session_2 starts a new EuroSim Mk3/Mk3rev1 simulator. Argument *session* is a structure containing all the information needed to launch a simulator.
Field *sim* specifies the simulator definition file (.sim). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
Field *work_dir* specifies the current working directory of the test controller. The directory should exist and be accessible by the EuroSim daemon. Normally this is done by making the directory available through NFS in case the RPC call is performed from a different host.
Field *simulator* is the file name of the simulator executable (.exe). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
schedule is the file name of the simulator schedule file (.sched). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
scenarios is an array of file names of scenario files (.mdl). It is possible to specify zero or more scenario files. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
dict is the file name of the data dictionary file (.dict). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
model is the file name of the model file (.model). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*. This file is not actually used by the simulator for reading. It used for tracing purposes as a reference.
recorderdir is the path name of the directory where all recordings are stored. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
initconds is an array of file names of initial condition files (.init). It is possible to specify zero or more initial condition files. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
exports is the file name of the exports file (.export). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
environment is an array of environment variables in the usual format VAR=value. Normally it is sufficient to copy the entire current environment into this array. If you want to start the simulator with a custom environment setting you have to set at least the following environment variables used by EuroSim in addition to the ones used by the simulator model software.
EFOROOT should be set to the EuroSim installation directory.
EFO_SHAREDMEMSIZE is the amount of memory reserved for dynamic memory allocation. Default is 4194304 (4 MB).
PWD is the current working directory and is set to *work_dir* by the daemon if it is not present.
LD_LIBRARYN32_PATH should be set to the path of the shared libraries of EuroSim for Irix 6.5. The value is normally $EFOROOT/lib32.
LD_LIBRARY_PATH should be set to the path of the shared libraries of EuroSim for other systems than Irix 6.5 (e.g. Linux). The value is normally $EFOROOT/lib.
prefcon is set to -1 under normal circumstances, a connection number is selected by the daemon and returned on successful start-up of the simulator. Put a positive value here if you want to force the new simulator to have a specific connection number.
realtime is set to 1 for real-time simulation runs, or 0 for non-real-time runs.
umask is the umask used for creation of new files. See **umask**(2).
The UNIX credentials passed as part of the request to start the simulator are used to deduce the uid and gid under which it will run. Make sure to create a valid authentication by calling *authunix_create_default*() (see **rpc**(3)).
The result is returned as a union with a selector variable *status*. If *status* equals **ST_SUCCESS**, the connection number can be retrieved from field *prefcon*. If *status* equals **ST_ERROR**, an list of error messages can be retrieved from field *errors*.

```
struct session2_def {
        file_def sim;
```

```
        char *work_dir;
        char *simulator;
        file_def schedule;
        struct {
                u_int scenarios_len;
                file_def *scenarios_val;
        } scenarios;
        char *dict;
        file_def model;
        char *recorderdir;
        struct {
                u_int initconds_len;
                file_def *initconds_val;
        } initconds;
        char *exports;
        struct {
                u_int environment_len;
                env_item *environment_val;
        } environment;
        int prefcon;
        int realtime;
        int umask;
};

struct start_result {
        int status;
        union {
                int prefcon;
                error_array errors;
        } start_result_u;
};
```

start_session_3 starts a new EuroSim Mk3rev2 simulator. Argument *session* is a structure containing all the information needed to launch a simulator.
Field *sim* specifies the simulator definition file (.sim). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
Field *work_dir* specifies the current working directory of the test controller. The directory should exist and be accessible by the EuroSim daemon. Normally this is done by making the directory available through NFS in case the RPC call is performed from a different host.
Field *simulator* is the file name of the simulator executable (.exe). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
schedule is the file name of the simulator schedule file (.sched). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
scenarios is an array of file names of scenario files (.mdl). It is possible to specify zero or more scenario files. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
dict is the file name of the data dictionary file (.dict). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
model is the file name of the model file (.model). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*. This file is not actually used by the simulator for reading. It used for tracing purposes as a reference.
recorderdir is the path name of the directory where all recordings are stored. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.

initconds is an array of file names of initial condition files (.init). It is possible to specify zero or more initial condition files. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
exports is the file name of the exports file (.export). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
environment is an array of environment variables in the usual format VAR=value. Normally it is sufficient to copy the entire current environment into this array. If you want to start the simulator with a custom environment setting you have to set at least the following environment variables used by EuroSim in addition to the ones used by the simulator model software.
EFOROOT should be set to the EuroSim installation directory.
PWD is the current working directory and is set to *work_dir* by the daemon if it is not present.
LD_LIBRARYN32_PATH should be set to the path of the shared libraries of EuroSim for Irix 6.5. The value is normally $EFOROOT/lib32.
LD_LIBRARY_PATH should be set to the path of the shared libraries of EuroSim for other systems than Irix 6.5 (e.g. Linux). The value is normally $EFOROOT/lib.
prefcon is set to -1 under normal circumstances, a connection number is selected by the daemon and returned on successful start-up of the simulator. Put a positive value here if you want to force the new simulator to have a specific connection number.
umask is the umask used for creation of new files. See **umask**(2).
flags is used to specify flags passed to the simulator at startup. Each bit of the variable is a flag. Currently two flags are specified:

SESSION_REALTIME

This flag set the real-time mode of the simulator if set.

SESSION_NO_AUTO_INIT

This flag prevents the EuroSim scheduler from automatically transferring to initialising state. This enables the client to set breakpoints and traces and disable tasks before the scheduler starts executing the tasks in initializing state.

The UNIX credentials passed as part of the request to start the simulator are used to deduce the uid and gid under which it will run. Make sure to create a valid authentication by calling *authunix_create_default*() (see **rpc**(3)).
The result is returned as a union with a selector variable *status*. If *status* equals **ST_SUCCESS**, the connection number can be retrieved from field *prefcon*. If *status* equals **ST_ERROR**, an list of error messages can be retrieved from field *errors*.

```
struct session3_def {
        file_def sim;
        char *work_dir;
        char *simulator;
        file_def schedule;
        struct {
                u_int scenarios_len;
                file_def *scenarios_val;
        } scenarios;
        char *dict;
        file_def model;
        char *recorderdir;
        struct {
                u_int initconds_len;
                file_def *initconds_val;
        } initconds;
        char *exports;
```

```
        struct {
                u_int environment_len;
                env_item *environment_val;
        } environment;
        int prefcon;
        int umask;
        int flags;
};

struct start_result {
        int status;
        union {
                int prefcon;
                error_array errors;
        } start_result_u;
};
```

start_session_4 starts a new EuroSim Mk4/Mk4rev1 simulator. Argument *session* is a structure containing all the information needed to launch a simulator.
Field *sim* specifies the simulator definition file (.sim). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
Field *work_dir* specifies the current working directory of the test controller. The directory should exist and be accessible by the EuroSim daemon. Normally this is done by making the directory available through NFS in case the RPC call is performed from a different host.
Field *simulator* is the file name of the simulator executable (.exe). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
schedule is the file name of the simulator schedule file (.sched). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
scenarios is an array of file names of scenario files (.mdl). It is possible to specify zero or more scenario files. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
dict is the file name of the data dictionary file (.dict). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
model is the file name of the model file (.model). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*. This file is not actually used by the simulator for reading. It used for tracing purposes as a reference.
recorderdir is the path name of the directory where all recordings are stored. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
initconds is an array of file names of initial condition files (.init). It is possible to specify zero or more initial condition files. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
calibrations is an array of file names of calibration files (.cal). It is possible to specify zero or more calibration files. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
exports is the file name of the exports file (.export). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
environment is an array of environment variables in the usual format VAR=value. Normally it is sufficient to copy the entire current environment into this array. If you want to start the simulator with a custom environment setting you have to set at least the following environment variables used by EuroSim in addition to the ones used by the simulator model software.
EFOROOT should be set to the EuroSim installation directory.
PWD is the current working directory and is set to *work_dir* by the daemon if it is not present.
LD_LIBRARYN32_PATH should be set to the path of the shared libraries of EuroSim for Irix 6.5. The value is normally $EFOROOT/lib32.

LD_LIBRARY_PATH should be set to the path of the shared libraries of EuroSim for other systems than Irix 6.5 (e.g. Linux). The value is normally $EFOROOT/lib.
prefcon is set to -1 under normal circumstances, a connection number is selected by the daemon and returned on successful start-up of the simulator. Put a positive value here if you want to force the new simulator to have a specific connection number.
umask is the umask used for creation of new files. See **umask**(2).
flags is used to specify flags passed to the simulator at startup. Each bit of the variable is a flag. Currently two flags are specified:

SESSION_REALTIME

This flag set the real-time mode of the simulator if set.

SESSION_NO_AUTO_INIT

This flag prevents the EuroSim scheduler from automatically transferring to initialising state. This enables the client to set breakpoints and traces and disable tasks before the scheduler starts executing the tasks in initializing state.

The UNIX credentials passed as part of the request to start the simulator are used to deduce the uid and gid under which it will run. Make sure to create a valid authentication by calling *authunix_create_default*() (see **rpc**(3)).
The result is returned as a union with a selector variable *status*. If *status* equals **ST_SUCCESS**, the connection number can be retrieved from field *prefcon*. If *status* equals **ST_ERROR**, an list of error messages can be retrieved from field *errors*.

```
struct session4_def {
        file_def sim;
        char *work_dir;
        char *simulator;
        file_def schedule;
        struct {
                u_int scenarios_len;
                file_def *scenarios_val;
        } scenarios;
        char *dict;
        file_def model;
        char *recorderdir;
        struct {
                u_int initconds_len;
                file_def *initconds_val;
        } initconds;
        struct {
                u_int calibrations_len;
                file_def *calibrations_val;
        } calibrations;
        char *exports;
        struct {
                u_int environment_len;
                env_item *environment_val;
        } environment;
        int prefcon;
        int umask;
        int flags;
};
```

```
struct start_result {
        int status;
        union {
                int prefcon;
                error_array errors;
        } start_result_u;
};
```

start_session_5 starts a new EuroSim Mk4rev1pl2/Mk4rev2 simulator. Argument *session* is a structure containing all the information needed to launch a simulator.
Field *sim* specifies the simulator definition file (.sim). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
Field *work_dir* specifies the current working directory of the test controller. The directory should exist and be accessible by the EuroSim daemon. Normally this is done by making the directory available through NFS in case the RPC call is performed from a different host.
Field *simulator* is the file name of the simulator executable (.exe). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
schedule is the file name of the simulator schedule file (.sched). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
scenarios is an array of file names of scenario files (.mdl). It is possible to specify zero or more scenario files. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
dict is the file name of the data dictionary file (.dict). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
model is the file name of the model file (.model). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*. This file is not actually used by the simulator for reading. It used for tracing purposes as a reference.
recorderdir is the path name of the directory where all recordings are stored. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
initconds is an array of file names of initial condition files (.init). It is possible to specify zero or more initial condition files. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
calibrations is an array of file names of calibration files (.cal). It is possible to specify zero or more calibration files. It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
exports is the file name of the exports file (.export). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
alias is the file name of the alias file (.alias). It can be an absolute or relative path name. If it is a relative path name, it is relative to the path in *work_dir*.
environment is an array of environment variables in the usual format VAR=value. Normally it is sufficient to copy the entire current environment into this array. If you want to start the simulator with a custom environment setting you have to set at least the following environment variables used by EuroSim in addition to the ones used by the simulator model software.
EFOROOT should be set to the EuroSim installation directory.
PWD is the current working directory and is set to *work_dir* by the daemon if it is not present.
LD_LIBRARYN32_PATH should be set to the path of the shared libraries of EuroSim for Irix 6.5. The value is normally $EFOROOT/lib32.
LD_LIBRARY_PATH should be set to the path of the shared libraries of EuroSim for other systems than Irix 6.5 (e.g. Linux). The value is normally $EFOROOT/lib.
prefcon is set to -1 under normal circumstances, a connection number is selected by the daemon and returned on successful start-up of the simulator. Put a positive value here if you want to force the new simulator to have a specific connection number.
umask is the umask used for creation of new files. See **umask**(2).

flags is used to specify flags passed to the simulator at startup. Each bit of the variable is a flag. Currently two flags are specified:

SESSION_REALTIME

This flag set the real-time mode of the simulator if set.

SESSION_NO_AUTO_INIT

This flag prevents the EuroSim scheduler from automatically transferring to initialising state. This enables the client to set breakpoints and traces and disable tasks before the scheduler starts executing the tasks in initializing state.

The UNIX credentials passed as part of the request to start the simulator are used to deduce the uid and gid under which it will run. Make sure to create a valid authentication by calling *authunix_create_default*() (see **rpc**(3)).
The result is returned as a union with a selector variable *status*. If *status* equals **ST_SUCCESS**, the connection number can be retrieved from field *prefcon*. If *status* equals **ST_ERROR**, an list of error messages can be retrieved from field *errors*.

```
struct session5_def {
        file_def sim;
        char *work_dir;
        char *simulator;
        file_def schedule;
        struct {
                u_int scenarios_len;
                file_def *scenarios_val;
        } scenarios;
        char *dict;
        file_def model;
        char *recorderdir;
        struct {
                u_int initconds_len;
                file_def *initconds_val;
        } initconds;
        struct {
                u_int calibrations_len;
                file_def *calibrations_val;
        } calibrations;
        char *exports;
        char *alias;
        struct {
                u_int environment_len;
                env_item *environment_val;
        } environment;
        int prefcon;
        int umask;
        int flags;
};

struct start_result {
        int status;
        union {
                int prefcon;
                error_array errors;
```

```
        } start_result_u;
};
```

kill_session_by_pid_1, **kill_session_by_pid_2**, **kill_session_by_pid_3**, **kill_session_by_pid_4** and **kill_session_by_pid_5** kills a simulator session identified by *pgid* with signal *signal*. Permissions to kill the simulator are checked by comparing the uid and gid in the credentials with those of the simulator. Root can always kill any simulator.
If the permissions were OK and the signal was sent successfully, the return value is 0. Otherwise, the return value is EPERM when you have not enough permission to kill the simulator, or ESRCH when there is no simulator with that *pgid*.

```
struct kill_pid {
        int pgid;
        int signal;
};
```

kill_session_by_name_1, **kill_session_by_name_2**, **kill_session_by_name_3**, **kill_session_by_name_4** and **kill_session_by_name_5** kills a simulator session identified by *name* with signal *signal*. Permissions to kill the simulator are checked by comparing the uid and gid in the credentials with those of the simulator. Root can always kill any simulator.
If the permissions were OK and the signal was sent successfully, the return value is 0. Otherwise, the return value is EPERM when you have not enough permission to kill the simulator, or ESRCH when there is no simulator with that *name*.

```
struct kill_name {
        char *name;
        int signal;
};
```

proc_allocate_1, **proc_allocate_2**, **proc_allocate_3**, **proc_allocate_4** and **proc_allocate_5** allocates real-time processors. The processor is removed from the pool of available processors. It is made isolated, non-preemptive and restricted. The processor number is returned. The isolated processor is added to the pool belonging to the simulator with pgid *pgid*.
rename_proc_by_pid_1, **rename_proc_by_pid_2**, **rename_proc_by_pid_3**, **rename_proc_by_pid_4** and **rename_proc_by_pid_5** was used to rename a process. This is now obsolete.

```
struct rename_pid {
        int pid;
        char *name;
};
```

check_license_1, **check_license_2**, **check_license_3**, **check_license_4** and **check_license_5** checks the EuroSim license on the machine. This is an internal call.

```
struct license_query {
        char *system_id;
        struct {
                u_int options_len;
                option_str *options_val;
        } options;
};
```

query_host2, **query_host3**, **query_host_4** and **query_host_5** query the host for its properties. The function returns the information in the following struct:


```
struct host_info {
       char *host_sysname;
       char *host_release;
       char *host_version;
       char *host_machine;
       int host_cpus;
};
typedef struct host_info host_info;
```

host_sysname is the operation system name of the host, e.g. IRIX, Linux.
host_release is the operating system release number, e.g. 6.5.20m, 2.4.20-18.timer.
host_version contains the date and time that the operating system was generated, e.g. 04091957, #1 Fri Sep 12 12:44:08 CEST 2003
host_machine contains the type of CPU board that the IRIX system is running on, e.g. IP27, i686
host_cpus is the number of CPU's of the host.

BUGS

Send bug reports to the EuroSim development team by using the spr tool, or mail them to esim-bugs@dutchspace.nl.

SEE ALSO

esimd(8), **esimd_complete_session**(3), **rpc**(3), **umask**(2).

EXAMPLES

To launch a simulator the following parameters could be used:

```
work_dir="/home/mt75377/EsimProjects/STD"
simulator="Demo.exe"
schedule="Demo.sched"
mission="Demo.mdl"
dict="Demo.dict"
model="Demo.model"
recorderdir="2000-02-29/15:55:11"
stimulatordir="."
initcond="Demo.init"
exports="Demo.exports"
alias="Demo.alias"
prefcon=-1
realtime=1
umask=022
```

The environment variables are for example:

```
LD_LIBRARY_PATH=/usr/EuroSim/lib
HOME=/home/mt75377
EFO_HOME=/home/mt75377/EsimProjects/EfoHome
LD_LIBRARYN32_PATH=/usr/EuroSim/lib32
PWD=/home/mt75377/EsimProjects/STD
EFOROOT=/usr/EuroSim
```

AUTHOR

Robert H. de Vries, Dutch Space BV, The Netherlands

2.2.15 esimd_client_launch, esimd_client_lauch_2, esimd_client_launch_3, esimd_client_launch_4, esimd_client_launch_5

Launch a EuroSim simulator

C SYNOPSIS

#include <esimd_clnt_launch.h>
cc ... -les
char *esimd_client_launch(const char *_hostname_, struct session_def *_session_);
char *esimd_client_launch_2(const char *_hostname_, struct session2_def *_session_);
char *esimd_client_launch_3(const char *_hostname_, struct session3_def *_session_);
char *esimd_client_launch_4(const char *_hostname_, struct session4_def *_session_);
char *esimd_client_launch_5(const char *_hostname_, struct session5_def *_session_, int *_con_, pid_t *_pid_);

DESCRIPTION

These functions all launch a EuroSim simulator using the RPC protocol of the EuroSim daemon. There are 5 versions of the protocol. Only the last one (**esimd_client_launch_5**) shall be used. The older ones are only provided for backwards compatibility.
The *hostname* is the name of the host on which the simulator shall run. The name can also be in numerical notation. The *session* contains the configuration of the simulator that will be started. This is normally filled in using **esimd_complete_session**().
The return value is NULL on success, otherwise an error string is returned. The string must be freed afterwards using **Free**(). The return value can also be NULL on an out of memory condition. In that case the value in *con (or session->prefcon for the old interfaces) will be -1.
The EuroSim connection number is returned in *con (or in session->prefcon) for the old interfaces on success.
The process ID of the simulator is returned in *pid on success.

SEE ALSO

auxmemory(3), **esimd_complete_session**(3)

AUTHOR

Robert de Vries, Dutch Space BV, The Netherlands

2.2.16 esimd_complete_session, esimd_update_arch, esimd_update_outputdir, esimd_free_session

EuroSim daemon RPC client interface helper functions

C SYNOPSIS

#include <esimd_complete_session.h>
cc ... -les
int esimd_complete_session(struct session5_def *_session_, const char *_hostname_, char **_env_);
void esimd_update_arch(struct session5_def *_session_, const char *_hostname_);
void esimd_update_outputdir(struct session5_def *_session_);
void esimd_free_session(struct session5_def *_session_);

DESCRIPTION

esimd_complete_session completes the *session* structure. All unfilled members of the structure are deduced from other members, most notably the *sim* member. If *sim* is set to the filename of a .sim file, then this file will be read and used to fill in the model, schedule, export file, alias file, scenarios and initial conditions if any of these are missing in the *session* structure.
If (possibly after reading the .sim file) either the model or the schedule file is missing in the *sim* structure, then error -1 is returned.
If the *work_dir* member is not set, then use **auxGetCwd**(3) to set it to a correct working directory.
If the *dict* member is not set, then use the model filename with the suffix .model replaced with suffix .dict. In order to use the correct system directory (e.g. Linux, IRIX64) the *hostname* must be set to the simulator host. If *hostname* is NULL, then the local host is used to determine the system name.
If the *simulator* member is not set, then use the model filename with the suffix .model replaced with suffix .exe. In order to use the correct system directory (e.g. Linux, IRIX64) the *hostname* must be set to the simulator host. If *hostname* is NULL, then the local host is used to determine the system name.
If the *recorderdir* member is not set, then create a directory name of the form YYYY-MM-DD/HH:MM:SS using the current time.
If the *environment.environment_val* member is not set, and the *env* argument is not NULL, then make a copy of the *env* environment.
prefcon is set to -1 and *umask* is set to the current umask value.
The SESSION_REALTIME flag in *flags* is set if the realtime flag is set to true in the simulation definition file.
This function returns 0 for success and -1 for failure.
esimd_update_arch updates the path names of the data dictionary and the executable in *session* to match the host type of the target machine *hostname*. This call must be performed after call **esimd_complete_session**().
esimd_update_outputdir updates the output directory for the result data. This is already done in **esimd_complete_session**(), but you must call this function again if you want to relaunch the simulator for a second time using the same session structure.
esimd_free_session frees all allocated memory of the *session* structure. This function assumes that all members where allocated using **auxmemory**(3).

SEE ALSO

auxmemory(3).

AUTHOR

Hans Verkuil, Dutch Space BV, The Netherlands

2.2.17 extClient

Functions for an external client to establish and control access to a EuroSim simulator

SYNOPSIS

#include <extSim.h>
cc ... -lesClient -les
typedef int (*EventHandler) (Connection *conn, const evEvent *command, void *data);
SIM *extConnect(const char * *host*, **const char *** *sysName*, **EventHandler** *clientHandler*, **void *** *data*, **unsigned int** *async*, **int** *prefcon*);
SIM *extConnectFd(const char * *host*, **const char *** *sysName*, **int** *fd*, **EventHandler** *clientHandler*, **void *** *data*, **unsigned int** *async*, **int** *prefcon*);
void extDisconnect(SIM * *sim*);
int extPoll(SIM * *sim*, **int** *block*);
int extPollTimeout(SIM * *sim*, **struct timeval *** *timeout*);
Connection *extSimConnection(const SIM * *sim*);
SIM *extSimId(Connection * *conn*);
int extInit(*void*);
int extExit(*void*);

DESCRIPTION

This library will enable an external client to connect to a EuroSim simulator and establish a loose coupling between the two simulations. From the moment the two simulations are coupled, the client will receive the state changes of the EuroSim simulator, and can get access to the EuroSim data dictionary (for reading and writing).
The external simulator access library is layered on top of eventConnection (see **evc**(3)).
The connection with a EuroSim simulator is initialized with a call to **extConnect**() or **extConnectFd**. The call will connect the client to a EuroSim simulator running at the given *host* with the given *prefcon* (preferred connection). The *fd* parameter of **extConnectFd** is the file descriptor returned by a successful **eventProbe**() call. The value of prefcon must be equal to the connection number automatically assigned to the simulator. This number can be found in two ways. The first method is by running the command line utility efoList. The connection number of your simulator can be found in the column with the header ”con”. The second method is by using the TestController. Select the menu option Simulation:Show Current Sims... At the end of the line you can find the connection number in the string ”(port <n>)” where <n> is the connection number. Connection numbers range from 0 to 9. The given argument *sysName* is the name under which this client will register itself at the EuroSim server. The given function pointer *clientHandler* is a call-back function (see also **evc**(3)) which will be activated as soon as events arrive from the network that cannot be handled within the module External Simulation Access like state-changes. The argument *data* is a pointer to user data which will be used as one of the parameters of the clientHandler call-back function. The argument *async* indicates whether the connection will be set-up asynchronous. If not, the user of the connection should use **extPoll**() or **extPollTimeout**() to poll for incoming events.
The function call **extDisconnect**() is used to close down the connection with the external EuroSim simulator. This function automatically destroys all associated views using **extViewDestroy**(). See **extView**(3).
The function call **extPoll**() is used to poll for incoming messages in the asynchronous mode. If *block* is set to true, the function will block until new data has arrived, otherwise the function will return immediately. This function will return 1 if any data was received, otherwise 0.
extPollTimeout() is similar to **extPoll**() with the only difference that it waits for the next incoming message until the timeout is expired.
extSimConnection() returns the associated **Connection*** for a extSim connection. (see **evc**(3).
extSimId() returns the associated **SIM*** for a Connection object.

extInit() initializes the external interface low-level API. This function only has meaning on Windows platforms, which require separate initialization of the RPC and Socket API. On the other platforms supported by EuroSim, this function always succeeds (and does nothing). Returns 0 when successful, -1 otherwise.
extExit() terminates the external interface low-level API. This function only has meaning on Windows platforms, which require separate termination of the RPC and Socket API. On the other platforms supported by EuroSim, this function always succeeds (and does nothing). Returns 0 when successful, -1 otherwise.

DIAGNOSTICS

All functions returning a pointer return a NULL pointer upon failure. Notice that since all requests end up in a request to the external EuroSim simulator, the only checks that are performed are about errors occurring before the request is pushed on the network. Any errors occurring afterward will not and cannot be reported.

SEE ALSO

evc(3), **extView**(3), **extExport**(3), **extRW**(8), **dict**(3).

BUGS

The value returned by a **EventHandler** is always ignored.
Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Erik de Rijk, Dutch Space BV, The Netherlands. Robert de Vries, Dutch Space BV, The Netherlands.

2.2.18 extView

Functions to create, control and destroy data views.

SYNOPSIS

#include <**extSim.h**>
client side
cc ... -lesClient -les
server side
cc ... -lesExt -lesServer -les
DView* extViewCreate(SIM* *sim*, **const char*** *id*);
int extViewAdd(DView* *view*, **const char*** *path*, **void*** *data*, **ExtVarType** *type*);
int extViewAddComplex(DView* *view*, **const char*** *path*, **void*** *data*, **const char*** *type*, **int** *ndims*, **const ExtArray*** *dims*, **int** *size*);
int extViewDelete(DView* *view*, **const char*** *path*);
int extViewByDict(DView* *view*, **DICT*** *dict*, **const char*** *path*, **const char*** *offset*, **int** *do_check*);
int extViewConnect(DView* *view*, **int** *flags*, **double** *frequency*, **int** *compression*);
int extViewChangeFrequency(DView* *view*, **double** *frequency*);
int extViewSend(DView* *view*);
DView* extViewId(const char* *channel*, **const char*** *id*);
int extViewDestroy(DView* *view*);

DESCRIPTION

This library will enable an external client to create and control views which can be used to retrieve or send data to a EuroSim simulator. Internally this library is also used by the EuroSim simulator to maintain its views.
A view is created with the call **extViewCreate**(). A **SIM** pointer is given to indicate which simulator should be the recipient of this view. At this point the external simulator is not yet notified about this new view. The second argument indicates the name of the view. This is the name with which the view is exported at the external EuroSim simulator (see **extExport**(3) for details).
There are two ways to add data-variables to a view. The first way is to manually add them.
The call **extViewAdd**() adds one variable to the view. The given path indicates where the variable is situated in the data dictionary of the external EuroSim simulator. This path is relative to the offset of the view (see **extExport**(3) for details). The data-pointer is the address where the incoming data will be stored, or where the outgoing data will be sought. This call is meant for the basic types. For the complex structures, **extViewAddComplex**() should be used.
The possible basic types are:

```
extVarType      C type
-------------------------------
extVarChar      char
extVarShort     short
extVarInt       int
extVarLong      long
extVarUnsChar   unsigned char
extVarUnsShort  unsigned short
extVarUnsInt    unsigned int
extVarUnsLong   unsigned long
extVarFloat     float
extVarDouble    double
```

The function **extViewAddComplex**() also has a path and an address where the data should be stored. With this function however, one can describe user defined types and arrays. The argument **type** is the

name of the type. This is either the name of the user defined type, or the name of the basic type. For the basic type, the following three defines have been made.

```
Type     C type
----------------------
STR_INT  char
         short
         int
         long
STR_UNS  unsigned char
         unsigned short
         unsigned int
         unsigned long
STR_FLT  float
         double
```

The type can also be generated out of a **Tnode** structure (See **dicttype**(3) for more information) with the function call **extType2str**().
The **ndims** and **dims** array are consistent with the use of arrays in the **DictAccess** structures. Ndims should be set to zero when there is only one element. The dims structure is then ignored. Ndims indicates how many ExtArray structures there are in the dims array. Each ExtArray structure is filled with a begin and an end from which the complete multi-dimensional array can be derived (See **dictaccess**(3) for more information). Check the extSim.h include file for the definition of *ExtArray*. The **size** is the total size of the complete structure, or array of structures.
If **ndims** is set to the special value **EXT_IGNORE_INDICES**, only the total size is checked and not the individual indices. This is useful if you want to specify an array at the client side with different dimensions than what is specified in the simulator model.
The second way to add variables is by including complete parts of data dictionaries. When one has a part of a data dictionary that matches the exported view of the external EuroSim simulator, that part can be added to the view. The function **extViewByDict**() will add all the variables that are found in the part of the given data dictionary, indicated by **path**. The branch of variables (entrypoints, comments and other dict-elements are ignored) must be an exact copy, or a subset, of the variables that are found in the part of the data dictionary at the external EuroSim simulator as indicated by **offset**. The given offset is relative to the offset of the given view.
The function **extViewDelete**() is used to delete elements from the view. The only useful purpose for this function is to erase elements from the view that are automatically added with the **extViewByDict**() call. It is not allowed to erase element when the view is already connected to the server.
When the view is created, the view can be registered at the external EuroSim simulator with a call to **extViewConnect**(). The given flags indicate which permissions must be granted for this view. The possible values are:

EXT_DICT_READ

Read-permission. Updates will be sent with a given frequency.

EXT_DICT_WRITE

Write-permission. Updates can be sent at will.

EXT_DICT_RW

Not yet supported. Imposes consistency problems (see **extRW**(8) for details).

The given frequency is only useful for views which are requested with **EXT_DICT_READ** permission. It indicates how many times per second a view must be sent to the client. This value must be more that zero. The frequency of the synchronous action manager is the highest frequency possible. Each tick, the synchronous actionmanager will distribute the views to the ones that need one. When the given

frequency is not a mean denominator of the frequency of the synchronous actionmanager, a smoothing algorithm is used to sent the exact requested number of updates per second. This frequency can later be changed with a call to **extViewChangeFrequency**().
The last argument indicates what compression should be used. For now only one compression method is available. There is no reason why one should not use this compression method. It simply discards values that are not changed since the last update and barely costs process time. The possible values are:

COMPRESS_NONE

No compression.

COMPRESS_CACHE

Suppress unchanged values.

When there are **no** changed values, the EuroSim server automatically discards the update. No matter whether compression is on or off.
When the view is connected, the updates from the server arrive automatically and they are processed automatically. When the view is opened with write permission, one can send an update to the EuroSim simulator with a call to **extViewSend**(). This will result in an event sent to the external EuroSim server, unless the data has not changed since the last call to **extViewSend**().
The incoming data from the server is processed automatically, but because the user should be notified that new data has arrived, the event is passed on through the given commandHandler that was given at the creation of the **SIM** pointer. One can find out which view this event was meant for, by extracting the first element from the event (which should be a string), and calling the function **extViewId**. The first argument should be the channel as set in the **evEvent** structure, and the second should be the retrieved string from the first argument.
When the view is not used anymore, it can be deleted by calling to **extViewDestroy**(). After this call it is still possible that data arrives from the server, but those updates are discarded. Do not call **extViewDestroy**() after calling **extDisconnect**() as it has already destroyed all views associated with that connection.

DIAGNOSTICS

All functions returning an **int** indicate errors with a negative value. A zero return indicates a successful call. All functions returning a pointer return a NULL pointer upon failure. Notice that since all request end up in a request to the external EuroSim simulator the only checks that are performed are about errors occurring before the request is pushed on the network. Any errors occurring afterward will and cannot be reported.

SEE ALSO

evc(3), **extExport**(3), **extRW**(8), **dict**(3). **dictaccess**(3).

BUGS

Read-write variables are not supported (see **extRW**(8) for details).
Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Erik de Rijk, Dutch Space BV, The Netherlands.

2.2.19 extMessage

Functions for an external client to send messages to a EuroSim simulator

SYNOPSIS

#include <extMessage.h>
cc ... -lesClient -les
void extMessage(SIM *sim, const char *format, ...);
void extWarning(SIM *sim, const char *format, ...);
void extError(SIM *sim, const char *format, ...);
void extFatal(SIM *sim, const char *format, ...);
void extReport(SIM *sim, esimSeverity s, const char *format, ...);

DESCRIPTION

The functions **extReport**, **extMessage**, **extWarning**, **extError** and **extFatal** can be used to send messages from a External Simulator Access client to a EuroSim simulator. The message will then be forwarded to the test-conductor interface and will also end up in the journal file. The first parameter *sim* is the pointer to the SIM structure returned by **extConnect**(). The **extReport** function allows the caller to specify the severity of the message. The other functions have implicit severities. The possible severity levels are:

esimSevMessage

comment or verbose information

esimSevWarning

warning messages

esimSevError

error messages

esimSevFatal

non-recoverable errors

The parameter *format* is a printf type format string and can be followed by arguments just like printf(). (see **printf**(3))

SEE ALSO

extClient(3), **esim**(3), **printf**(3).

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert de Vries, Dutch Space BV, The Netherlands.

2.2.20 extMdl

Functions for an external client to manage missions and actions running on a EuroSim simulator

SYNOPSIS

#include <**extMdl.h**>
cc ... -lesClient -les
void extMissionNew(SIM ****sim*, const char ****mdl_file***);**
void extMissionOpen(SIM ****sim*, const char ****mdl_file***);**
void extMissionClose(SIM ****sim*, const char ****mdl_file***);**
void extActionNew(SIM ****sim*, const char ****mdl_file***, const char ****action_script***);**
void extActionDelete(SIM ****sim*, const char* ****mdl_file***, const char ****action_name***);**
void extActionExecute(SIM ****sim*, const char ****mdl_file***, const char ****action_name***);**

DESCRIPTION

These functions can be used to manage missions and actions on a running EuroSim simulator. Before these functions can be used, a connection has to be created using **extConnect**() (see **extClient**(3)). The first parameter of all functions is the reference to the *SIM** returned by **extConnect**(). On invocation, these functions will result in a message on the test controller screen. If an error occurs an error message is sent to the test controller screen.
The function **extMissionNew**() creates a new mission file with the name *mdl_file*. This mission exists only in the running simulator. It can be used as a temporary action container. The name of the file must end with ".mdl" even though the mission is not in an actual file. The function **extMissionOpen**() opens the existing mission *mdl_file* on disk. The function **extMissionClose**() closes mission *mdl_file* and removes it from the simulator altogether.
The function **extActionNew**() adds a new action to mission file *mdl_file*. The script *action_script* contains among other things the name of the action. This name can be used later to identify the action. The function **extActionDelete**() deletes from mission *mdl_file* the action *action_name.* The function **extActionExecute**() executes the action from mission *mdl_file* named *action_name* once.

SEE ALSO

extClient(3).

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert de Vries, Dutch Space BV, The Netherlands.

2.2.21 extSync

Functions for an external client to synchronize with a EuroSim simulator

SYNOPSIS

CLIENT-SIDE

#include <extSim.h>
cc ... -lesClient -les
int extSyncSend(const SIM *_sim_, int _token_);
int extSyncRecv(const SIM *_sim_, int _token_);

SERVER-SIDE

#include <esimExt.h>
int esimExtSyncSend(int _token_);
int esimExtSyncRecv(int _token_);

DESCRIPTION

These functions allow one or two-way synchronization of external applications with a EuroSim simulator via the network. The synchronization requires the addition of two function calls on each side of the interface in order to synchronize the two applications. The synchronization requires the simulator to run in non-realtime mode as the synchronization introduces delays which are incompatible with hard real-time predictable execution timings. When the two applications are synchronized in both directions, each application will stop when the other stops. When the application continues again the other will continue as well. Also small variations in execution speed can be handled this way.
In order to get the desired result, the calls must be performed in a specific order. First the **extSyncSend**(client) or **esimExtSyncSend**(model) call is done, followed by an **extSyncRecv**(client) or **esimExtSyncRecv**(model) call. The sending of the sync should follow the sending of an external simulator access update if the synchronization is used to make sure that data exchanged between the client and the simulator is updated at the receiving end. At the client side, the update is done by a call to **extViewSend**. At the simulator side, the update is done implicitly in the call **esimExtSyncSend**.
A call to the function **extSyncRecv** or **esimExtSyncRecv** can block.
esimExtSyncSend shall only be called when the client for which the token is intended is connected. If this is not the case the token is lost and the synchronization mechanism breaks down. The proper order is therefore to start the simulator first. Place the simulator in a stable state (stand-by state for example) where no synchronization takes place. After the client has connected, the simulator state can be switched to executing state where the synchronization takes place.
The *sim* argument (client side only) is the connection to be used in the synchronization.
The *token* argument is the token identifier. The identifier should be unique for each receiving side (client and simulator). If the simulator has multiple clients, the tokens sent to the simulator from each client shall be unique. The *token* is used to differentiate between different synchronization points in the simulator. Token numbers shall be unique for each synchronization point.

DIAGNOSTICS

If the function succeeds it returns 0, if it fails it returns -1.

SEE ALSO

extClient(3).

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert de Vries, Dutch Space BV, The Netherlands.

2.2.22 EuroSim

EuroSim interactive shell functions.

SYNOPSIS

use EuroSim;

DESCRIPTION

EuroSim interactive shell functions

setup_readline

Setup the command line interface handler.

help

print help message.

print_host_list

Print the list of EuroSim hosts.

print_session_list HOSTNAME

Print the EuroSim simulators running on host *HOSTNAME*. The default host is the localhost. If you use the magic hostname **all**, all sessions on the subnet are shown.

log_open FILENAME

Open a log file to store all given commands. The resulting file is a perl script including a header importing the EuroSim modules.

log_close

Close the log file. Commands are no longer saved.

AUTHOR

Robert H. de Vries, Dutch Space BV

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

SEE ALSO

perl(1), EuroSim::Session(3), EuroSim::Conn(3), EuroSim::Dict(3), EuroSim::Link(3), EuroSim::InitCond(3), EuroSim::MDL(3), EuroSim::SimDef(3).

2.2.23 EuroSim::Session

EuroSim Test Controller functions

SYNOPSIS

use EuroSim::Session;

DESCRIPTION

OBJECT ORIENTED INTERFACE

This module has an object oriented interface. The central object is the **SESSION**. New sessions are created by starting a new simulator or connecting to an already running simulator. To start a new simulator see STARTING A SIMULATOR. To connect to a simulator use `esim_connect SESSION`. If only one simulation is managed by the application it is not necessary to call the methods as follows:

```
$s->go;
```

The only thing you need to do is call:

```
go;
```

This is mainly done to reduce the amount of typing in interactive mode. This means that for most functions below the first parameter is optional. In the `$s->go;` notation above the variable `$s` is passed as the first parameter. If the first parameter is not a session the current active session is used. If there is no session an error message is printed.

DEFAULT EVENT HANDLERS

The following events are handled automatically:

- maDenyWriteAccess
- maCurrentWorkingDir
- maCurrentAliasFile
- maCurrentTSPMapFile
- maCurrentCycletime
- maCurrentTimeMode
- maSimDef
- maNewMission
- maCurrentInitconds
- maCurrentCalibrations
- maRecording
- scGoRT
- scTaskListStart, scTaskStart, scTaskEntry, scTaskEnd, scTaskListEnd
- scEventListStart, scEventInfo, scEventListEnd
- scWhereList, scWhereEntry, scWhereListEnd

- scSetBrk
- scSetTrc
- scTaskDisable
- scStepTsk
- scContinue
- scSpeed
- dtHeartBeat
- maRecordingBandwidth
- maStimulatorBandwidth
- rtUnconfigured, rtInitialising, rtStandby, rtExecuting, rtExiting
- rtMainCycle
- dtMonitor, dtMonitorVar, dtMonitorDone
- dtLogValueUpdate

USER DEFINED EVENT HANDLERS

It is also possible to add your own event handlers. Adding a new event handler is done by calling:

```
$s->event_addhandler("rtExecuting", \&handle_executing);
```

The callback is called with the following arguments:

1. Session object, reference to the session hash
2. Name of the event (the enumeration identifier name)
3. Simulation time (sec)
4. Simulation time (nsec)
5. Wall-clock time (sec)
6. Wall-clock time (nsec)
7. Parameters (event specific)

Example:

```
sub cb_standby
{
    my ($session, $event_name, $simtime_sec, $simtime_nsec,
            $wallclock_sec, $wallclock_nsec) = @_;
    print "going to standby at $wallclock_sec\n";
}

$session->event_addhandler("rtStandby", \&cb_standby);

# or a bit more compact
$session->event_addhandler("rtExecuting",
    sub { print "going to executing at $_[4]\n"; });
```

STARTING A SIMULATOR

To start a realtime simulator all you need to do is:

```
$s = new EuroSim::Session("some.sim");
$s->realtime(1);
$s->init;
```

This command will use the information defined in the simulation definition file to start the simulator. The realtime flag results in a real-time run of the simulator.
After a successful simulator launch with init, the function automatically connects itself to the simulator. This is followed by subscription to all channels. Various default event handlers are installed to handle the messages sent autonomously through the channels. The information is stored in the session structure and can be printed by calling `print_session_parameters`. When running in interactive mode, a journal window is created showing all messages received from the simulator.
The incoming messages are stored as follows into the session structure in the form of hash fields. For example the initial conditions are stored in the field initconds and can be retrieved with $s->{initconds}. For more information on the session datastructure see `new`.

EuroSim::Session Functions

new SIM HOSTNAME

new SIM

new

Create a new session object. If you pass a *SIM* filename the simulation definition file and the MDL files it refers to is loaded ready for starting. If the loading fails `undef` is returned. You can also specify the hostname if the simulation does not run on the local host. The Session object is a hash structure with the following fields:

MDL

Hash table of loaded MDL files. Each hash key is the name of a loaded MDL file. The hash value is a EuroSim::MDL object. MDL files are loaded at start-up when a .sim file is loaded or during run-time when extra MDL files are loaded. Extra files can be loaded by the event handler for event `maNewMission` or by manually adding MDL files with new_scenario.

clientname

The name under which this session is known to the simulator. The value is set with the function `clientname`.

conn

EuroSim::Conn object. Low level connection object.

cwd

Current working directory of the simulator. The value is set by the event handler for event `maCurrentWorkingDir`.

dict

Data dictionary file name. The value is set by the event handler for event `maCurrentDict`.

eventlist

List of events present in the schedule. The value is set by the event handler for the following events: `scEventListStart`, `scEventInfo`, `scEventListEnd`. The eventlist is an array of hash tables. Each table consists of three elements:

name

The name of the event.

state
The scheduler state for which it is defined.

is_standard
Flag indicating that it is a standard event, i.e. predefined by EuroSim.

handler
Event handler table.

sim_hostname
Simulation host name. The value is set with the function `sim_hostname`.

startup_timeout
Simulation startup timeout. The default value is 5 seconds and it can be change with the function `startup_timeout`.

initconds
Initial condition files. The value is set by the event handler for event `maCurrentInitconds`.

calibrations
Calibration files. The value is set by the event handler for event `maCurrentCalibrations`.

logwindow
EuroSim::Window object. Used to display test controller messages in interactive mode.

monitored_vars
Table of monitored variables.

outputdir
Result directory used in current simulation run. The value is set by the event handler for event `maCurrentResultDir`.

prefcon
Connection number.

realtime
Realtime mode. 1 is real-time, 0 is non-realtime. The value is set by the event handler for event `scGoRT`.

recording
Flag indicating that recording is enabled or not. 1 means enabled. 0 means disabled. The value is set by the event handler for event `maRecording`.

recording_bandwidth
Recorder bandwidth in bytes/second. The value is set by the event handler for event `maRecordingBan`

schedule
Schedule file name. The value is derived from the model file name when you set the scenario.

simdef
Simulation definition handle to a `EuroSim::SimDef` object.

sim_time
The simulation time (as seen by the running simulator). The value is set by the event handler;

speed
The clock acceleration factor achieved by the simulator. Values larger than 1 indicate faster than real-time. Values smaller than 1 indicate slower than real-time. The value is set by the event handler for event `scSpeed`.

state
Simulator state. Can be: unconfigured, initialising, stand-by, executing, exiting. The value is set by the event handler for the following events: `rtUnconfigured`, `rtInitialising`, `rtStandby`, `rtExecuting` and `rtExiting`.

stimulator_bandwidth

Stimulator bandwidth in bytes/second. The value is set by the event handler for event `maStimulatorBandwidth`.

tasklist

List of tasks present in the schedule. The value is set by the event handler for event `scTaskListStart`, `scTaskStart`, `scTaskEntry`, `scTaskEnd` and `scTaskListend`. The field tasklist is a hash table. Each key in the hash table is the name of a task (e.g. `$session->{tasklist}->{taskname`
Each task consists of a number of entry points and a flag called disable. The disable flag is set by the event handler of `scTaskDisable`. The entry points are stored in an array. Each array element is a hash table consisting of three fields:

name

The name of the entry point.

breakpoint

Flag indicating that a breakpoint has been set on this entry point. The value is set by the event handler for event `scSetBrk`.

trace

Flag indicating that this entry point is being traced. The value is set by the event handler for event `scSetTrc`.

time_mode

The time mode can be relative or absolute (UTC). Relative is 0 and absolute is 1. The value is set by the event handler for event `maCurrentTimeMode`.

workdir

Work directory used in current simulation run. If not set, then the current directory is used.

alias

Alias file used in current simulation run. The value is set by the built-in event handler of event `maCurrentAliasFile`.

tsp_map

TSP map file used in current simulation run. The value is set by the built-in event handler of event `maCurrentTSPMapFile`.

user_defined_outputdir

User defined output directory path. This directory path overrides the default output directory path. The value is set with the function `outputdir`.

wallclock_time

The wallclock time (as seen by the running simulator). The value is set by the event handler;

wallclock_boundary

The wallclock boundary time to be used for timed state transitions. If you add an integer number of times the main_cycle time to this value it will produce a valid state transition boundary time.

simtime_boundary

The simulation time boundary to be used for timed state transitions. If you add an integer number of times the main_cycle time to this value it will produce a valid state transition boundary time.

main_cycle

The main cycle time of the current schedule. It can be used to calculate valid boundary times for timed state transitions.

watcher

Event::io object. Used to process incoming events.

where

Current breakpoint. The value is set by the event handlers for the following events: `scWhereListStar` `scWhereEntry`, `scWhereListEnd`. It is cleared by the following events: `scStepTsk` and `scContinue`. The value is an array of value pairs stored in an array. The first value in the array is the task name and the second is the entry number.

write_access

Flag to indicate whether this client is allowed to change variable values in the simulator. The value is set by the event handler for event `maDenyWriteAccess`.

simdef SESSION,FILENAME,HOSTNAME

simdef SESSION,FILENAME

simdef SESSION

Set the simulation definition file of the session. If *FILENAME* is passed to the function, the simulation definition filename is set to the new value. If the simulation does not run on the local host then the *hostname* should also be specified. This simulation definition file is used to start the simulator. Information derived from the simulation definition file is used to provided sensible defaults for all parameters. See EuroSim::SimDef(3).

dict SESSION

Return the current data dictionary.

realtime SESSION,FLAG

realtime SESSION

Return flag to indicate if the simulation is running real-time or not. If *FLAG* is passed to the function the flag is set to the new value. If *FLAG* is set to 1 realtime is enabled. If *FLAG* is set to 0 the run will be non-realtime. Non-realtime is the default.

auto_init SESSION,FLAG

auto_init SESSION

Return flag to indicate if the simulation performs automatic state transition to initializing or not. If *FLAG* is passed to the function the flag is set to the new value. If *FLAG* is set to 1 automatic state transition to initializing is enabled. If *FLAG* is set to 0 the simulator will wait in unconfigured state until an explicit state transition to initializing is commanded. Automatic state transition to initializing is the default.

sim_hostname SESSION,HOSTNAME

sim_hostname SESSION

Return hostname where the current simulation session is running. If *HOSTNAME* is passed as an argument to the function, the hostname is set. The hostname will be used for future simulation runs.

prefcon SESSION,PREFCON

prefcon SESSION

Return the connection number of the current simulation session. If *PREFCON* is passed as an argument to the function, the connection number is set. This can be used in a situation where you need to reconnect to an already running simulator. To start new simulation runs, this number is not used.

workdir SESSION,WORKDIR

workdir SESSION

Return the work directory name of the current simulation session. If *WORKDIR* is passed as an argument to the function, the work directory is set for the next run. This directory is where all input files are found and where the output directory is created. By default the directory is set to the current directory at the moment of starting a new simulation.

outputdir SESSION,OUTPUTDIR

outputdir SESSION

Return the output directory name of the current simulation session. If *OUTPUTDIR* is passed as an argument to the function, the output directory is set for the next run. This directory is where all output files will be stored. Output files are recorder files, snapshot files, the timing file and the journal file. By default the directory is set to *date*/*time* at the moment of starting a new simulation.

alias SESSION,ALIAS

alias SESSION

Return the alias file name of the current simulation session. If *ALIAS* is passed as an argument to the function, the alias file is set for the next run. The alias file is used to create aliases of data dictionary variables.

tsp_map SESSION,TSP_MAP

tsp_map SESSION

Return the TSP map file name of the current simulation session. If *TSP_MAP* is passed as an argument to the function, the TSP map file is set for the next run. The TSP map file is used to define the variables exported by the TSP provider in EuroSim.

journal SESSION

Return the journal file name of the current simulation session. The journal file name is *date*/*time*/EsimJournal.txt. Where *date* and *time* are equal to the time of starting the simulator.

clientname SESSION,CLIENTNAME

clientname SESSION

Return the client name of the current simulation session. If *CLIENTNAME* is passed as an argument to the function, the client name is set for the next run. The client name is the name under which the application calling this function will be known to the simulator. By default this is ”esimbatch”.

startup_timeout SESSION,TIMEOUT

startup_timeout SESSION

Return the startup timeout in seconds of the current simulation session. If *TIMEOUT* is passed as an argument to the function, the startup timeout is set. The startup timeout default is 5 seconds. If starting up a simulator takes longer than this you must change that default to a higher value.

initconds SESSION,INITCONDS

initconds SESSION

Override the initial condition file(s) specified in the SIM file. INITCONDS is a reference to an array of initial condition file names. If no INITCONDS parameter is passed it only returns the an array with the current initial condition file names.

calibrations SESSION,CALIBRATIONS

calibrations SESSION

Override the calibration file(s) specified in the SIM file. CALIBRATIONS is a reference to an array of calibration file names. If no CALIBRATIONS parameter is passed it only returns the an array with the current calibration file names.

scenarios SESSION,SCENARIOS

scenarios SESSION

Override the MDL file(s) specified in the SIM file. SCENARIOS is a reference to an array of MDL file names. If no SCENARIOS parameter is passed it only returns the an array with the current MDL file names.

event_addhandler SESSION,EVENT_NAME,HANDLER

event_addhandler EVENT_NAME,HANDLER

Add event handler *HANDLER* to handle event *EVENT_NAME*. The event handler is added to the end of the list if there are already event handlers installed for this event. The installed handler is returned for later use with event_removehandler.

event_removehandler SESSION,EVENT_NAME,HANDLER

event_removehandler EVENT_NAME,HANDLER

Remove callback function *HANDLER* for event with name *EVENT_NAME*.

sev_to_string SESSION, SEV

Return string representation of severity SEV.

esim_connect SESSION

Connect to a running simulator as specified in *SESSION*. A new journal file is opened and a window with test controller messages is opened in interactive mode.

init SESSION

init

Start a new simulation run.

finish SESSION

finish

Finish a stopped simulation session. When a simulation is run in interactive mode a window appears with the test controller messages. When the simulation stops, this window is not removed to make it possible to read the final messages. In order to finally close that window too, call this function.

esim_disconnect SESSION

esim_disconnect

Disconnect from the simulation session *SESSION*. The simulator will continue to run in the background.

join_channel SESSION,CHANNEL

join_channel CHANNEL

Join a channel of a simulation session. By default each session connects to all channels. The following channels are available: mdlAndActions, data-monitor, rt-control, sched-control. To join all channels use channel ”all”.

leave_channel SESSION,CHANNEL

leave_channel CHANNEL

Leave a channel of a simulation channel.

wait_event SESSION,EVENT_NAME,TIMEOUT

wait_event SESSION,EVENT_NAME

wait_event EVENT_NAME,TIMEOUT

wait_event EVENT_NAME

Synchronously wait for an event with name *EVENT_NAME*. The default timeout is 10 seconds. Other events are processed normally. If the event arrived within the timeout period then the name of that event is returned.

wait_time SESSION,TIMEOUT

wait_time TIMEOUT

Synchronously wait for *TIMEOUT* seconds. Other events are processed normally during that time.

quit

Quit the batch utility. Stops current simulation run.

print_monitored_vars SESSION

print_monitored_vars

Print a list of currently monitored variables and their current values. All variables in active monitors send values to the batch tool. A table with all variables is kept with their current values.

print_session_parameters SESSION

print_session_parameters

Print a complete overview of all parameters available in the SESSION object. The SESSION object is a blessed reference to a hash table containing all data relevant for the simulation session. This function prints a human readable presentation of that hash table.

go SESSION,SEC,NSEC

go SEC,NSEC

go SESSION

go

Change the simulator state from stand-by to executing. Equivalent to the Go button of the test controller. The variant specifying the time is used for timed state transitions. The wallclock time is secified as *SEC* seconds and *NSEC* nanoseconds.

stop SESSION,SEC,NSEC

stop SEC,NSEC

stop SESSION

stop

Stop the simulation run. Equivalent to the Stop button of the test controller. The variant specifying the time is used for timed state transitions. The wallclock time is secified as *SEC* seconds and *NSEC* nanoseconds.

freeze SESSION,SEC,NSEC

freeze SEC,NSEC

freeze SESSION

freeze

pause SESSION,SEC,NSEC

pause SEC,NSEC

pause SESSION

pause

Change the simulator state from executing to stand-by. Equivalent to the Pause button of the test controller. The variant specifying the time is used for timed state transitions. The wallclock time is secified as *SEC* seconds and *NSEC* nanoseconds.

freeze_at_simtime SESSION,SEC,NSEC

freeze_at_simtime SEC,NSEC

Change the simulator state from executing to stand-by on the specified simulation time. The simulation time is secified as *SEC* seconds and *NSEC* nanoseconds.

step SESSION,SEC,NSEC

step SEC,NSEC

step SESSION

step

Run the simulation models for one main scheduler cycle or as long as specified. Equivalent to the Step button of the test controller when no duration is specified.

abort SESSION

abort

Abort the current simulation run. Equivalent to the Abort button of the test controller.

health SESSION

health

Request a health check of the running simulator. Prints health information to the test controller window.

reset_sim SESSION

reset_sim

Restart the current simulation with the current settings. Equivalent to the Reset button of the test controller.

new_scenario SESSION,SCENARIO

new_scenario SCENARIO

Create a new scenario in the simulator. This new scenario is only a container for new actions. It is not a file on disk. It is a pure in core representation.

open_scenario SESSION,SCENARIO

open_scenario SCENARIO

Open a new scenario file in the simulator with file name *SCENARIO*. The file must be on disk and readable. The function returns the handle to the newly loaded MDL file or undef if the file cannot be loaded.

close_scenario SESSION,SCENARIO

close_scenario SCENARIO

Close a currently opened scenario with name *SCENARIO* in the simulator.

new_action SESSION,SCENARIO,ACTION

new_action SCENARIO,ACTION

Add a new action in the scenario file with name *SCENARIO*. *ACTION* is the complete action text. There are a few utility functions to generate those actions. See EuroSim::MDL(3).

delete_action SESSION,SCENARIO,ACTION

delete_action SCENARIO,ACTION

Delete an action from scenario *SCENARIO* with name *ACTION*.

action_execute SESSION,SCENARIO,ACTION

action_execute SCENARIO,ACTION

Trigger the execution of the action with name *ACTION* in scenario with name *SCENARIO*. This is equivalent to triggering an action manually on the scenario canvas of the Simulation Controller.

action_activate SESSION,SCENARIO,ACTION

action_activate SCENARIO,ACTION

Make action with name *ACTION* in scenario with name *SCENARIO* active in the running simulator. The action must already be defined in the scenario. This is equivalent to ctivating an action on the scenario canvas of the Simulation Controller.

action_deactivate SESSION,SCENARIO,ACTION

action_deactivate SCENARIO,ACTION

Deactivate action with name *ACTION* in scenario with name *SCENARIO* in the running simulator. This is equivalent to deactivating an action on the scenario canvas of the Simulation Controller.

execute_command SESSION,NAME,COMMAND,ACTION_MGR_NR

execute_command SESSION,NAME,COMMAND

execute_command NAME,COMMAND,ACTION_MGR_NR

execute_command NAME,COMMAND

Execute command *COMMAND* with name *NAME* on action mgr *ACTION_MGR_NR* in the running simulator. This is the equivalent of creating an action, executing it and deleting it again.

snapshot SESSION,FILENAME,COMMENT

snapshot SESSION,FILENAME

snapshot SESSION

snapshot FILENAME,COMMENT

snapshot FILENAME

snapshot

Make a snapshot of the current state of the variables in the data dictionary. The *COMMENT* string is optional. If you omit the filename, a filename is chosen of the form snapshot_<simtime>.snap. The snapshot is saved in the output directory, unless the filename is absolute. This is equivalent to the "Take Snaphot..." menu option in the "Simulation" menu of the test controller.

mark SESSION,COMMENT

mark SESSION

mark COMMENT

mark

Make a mark in the journal file. The *COMMENT* string is optional. This is equivalent to the "Make Mark" and "Make Comment" menu options in the "Journal" menu of the Test Controller.

sim_debug SESSION,MESSAGE

sim_debug MESSAGE

Send a debugging message to the simulator. Only used internally for debugging.

sim_message SESSION,MESSAGE

sim_message MESSAGE

Send a message to the simulator for distribution to all clients. This is useful if your client application is not the only client of the simulator. The message is broadcasted to all clients.

suspend_recording SESSION

suspend_recording

Suspend recording in the simulator. This is equivalent to using the "Suspend Recording" menu item of the "Simulation" menu of the test controller. Use

resume_recording SESSION

resume_recording

Resume recording in the simulator. This is equivalent to using the "Resume Recording" menu item of the "Simulation" menu of the test controller.

recording_switch SESSION

recording_switch

Switch all recording files of a simulation run. All currently open recorder files are closed and new recorder files are created. Recording will continue in the new recorder files.

set_initcond SESSION,INITCONDS

set_initcond INITCONDS

Set the initial condition file of the simulation to *INITCONDS*. You have to do a reset on the simulator before these new initial conditions are taken into account.

For example:

```
$s->set_initcond("a.init", "b.init", "c.init");
$s->reset_sim;
```

set_calibration SESSION,CALIBRATIONS

set_calibration CALIBRATIONS

Set the initial condition file of the simulation to *CALIBRATIONS*. You have to do a reset on the simulator before these new initial conditions are taken into account.

For example:

```
$s->set_calibration("a.cal", "b.cal", "c.cal");
$s->reset_sim;
```

reload SESSION,SNAPFILE,HARD

reload SESSION,SNAPFILE

reload SNAPFILE,HARD

reload SNAPFILE

Load initial condition file or snapshot file with file name *SNAPFILE* into the running simulator. If parameter *HARD* is by default 0. This means that the simulation time stored in the snapshot file is ignored. If *HARD* is set to 1, the simulation time is set to the value specified in the snapshot file.

monitor_add SESSION,VARNAME

monitor_add VARNAME

Start monitoring variable *VARNAME*. The value of the variable is updated with 2 Hz. The function `print_monitored_vars` prints an overview of all monitored variables.

monitor_remove SESSION,VARNAME

monitor_remove VARNAME

Stop monitoring a variable *VARNAME*.

monitor_set SESSION,VARNAME,VALUE

monitor_set VARNAME,VALUE

Set variable *VARNAME* to value *VALUE* in the simulator. A message is printed in the journal file for tracing purposes.

monitor_set_multi SESSION,VAR_VALUE_HASH

monitor_set_multi VAR_VALUE_HASH

Set multiple variables defined in the *VAR_VALUE_HASH* to the specified value in the simulator. The key of each hash element is the name of the variable and the value is the new value of the variable. A message is printed in the journal file for tracing purposes.

monitor_get SESSION,VARNAME

monitor_get VARNAME

Get value of variable *VARNAME* from the simulator.

cpuload_set_peak SESSION,PROCESSOR,PEAK_TIME

cpuload_set_peak PROCESSOR,PEAK_TIME

Configure the CPU load monitor peak time in msecs.

set_breakpoint SESSION,TASKNAME,ENTRYNR,ENABLE

set_breakpoint TASKNAME,ENTRYNR,ENABLE

Set a breakpoint on entry nr *ENTRYNR* in task *TASKNAME* in the scheduler. If parameter *ENABLE* is set to 1 the breakpoint is enabled. To disable it again set the parameter to 0.

set_trace SESSION,TASKNAME,ENTRYNR,ENABLE

set_trace TASKNAME,ENTRYNR,ENABLE

Enable/disable tracing of entry points. Entry points are defined by specifying the number of the entry point *ENTRYNR* (numbering starts at 0) and the name of the task *TASKNAME*. To enable a trace set *ENABLE* to 1, to disable it set it to 0. Tracing an entry point means that messages are printed to the journal window.

where SESSION

where

Request the current position when the scheduler has stopped on a break point. The reply to the message is automatically stored in the SESSION structure. Normally the position is sent to the client whenever the scheduler hits a breakpoint. So there is rarely any need to request the position manually if you store the position on the client side (as is done in this tool.)

step_task SESSION

step_task

Perform one step (=one entrypoint) in the scheduler debugger.

cont SESSION

cont

Continue executing upto the next breakpoint in the scheduler debugger.

list_tasks SESSION

list_tasks

Request a list of all tasks in the current schedule of the simulator. The list is also sent automatically upon joining the ”sched-control” channel. The information is stored in the SESSION structure.

task_disable SESSION,TASKNAME

task_disable TASKNAME

Disable task with name *TASKNAME* in the current schedule of the simulator.

task_enable SESSION,TASKNAME

task_enable TASKNAME

Enable task with name *TASKNAME* in the current schedule of the simulator.

clear_breaks SESSION

clear_breaks

Remove all breakpoints in the current schedule of the simulator.

clear_traces SESSION

clear_traces

Remove all traces in the current schedule of the simulator.

set_simtime SESSION,SIMTIME_SEC,SIMTIME_NSEC

set_simtime SIMTIME_SEC,SIMTIME_NSEC

Set the simulation time to *SIMTIME_SEC* seconds and *SIMTIME_NSEC* nanoseconds. This can only be done in stand-by state.

enable_realtime SESSION

enable_realtime

Switch to real-time mode. This can only be done when the simulator has started off in real-time mode, and has switched to non-real-time mode.

disable_realtime SESSION

disable_realtime

Switch to non-real-time mode.

list_events SESSION

list_events

Request a list of all events in the schedule of the simulator in all states. The result is stored in the SESSION structure. The list is automatically sent to the client when subscribing to the ”sched-control” channel at start-up.

raise_event SESSION,EVENT

raise_event EVENT

Raise event with name *EVENT* in the scheduler. An event is defined by the input connector on the scheduler canvas. The event is handled as fast as possible.

raise_event_at SESSION,EVENT,SEC,NSEC

raise_event_at EVENT,SEC,NSEC

Raise event with name *EVENT* in the schedler at a specified wallclock time. The wallclock time is specified as *SEC* seconds and *NSEC* nanoseconds.

raise_event_at_simtime SESSION,EVENT,SEC,NSEC

raise_event_at_simtime EVENT,SEC,NSEC

Raise event with name *EVENT* in the schedler at a specified simulation time. The simulation time is specified as *SEC* seconds and *NSEC* nanoseconds.

speed SESSION,ACCELERATION

speed ACCELERATION

Set the acceleration/deceleration of the scheduler of the simulator. Values smaller than 1 will cause a proportional deceleration of the scheduler clock. Values larger than 1 will cause a proportional acceleration of the scheduler clock. Magical value -1 means that the scheduler will run in an optimized as-fast-as-possible mode.

entrypoint_set_enabled SESSION,ENTRYPOINTNAME,ENABLED

Enable/disable entry point. If ENABLED is 1 the entry point ENTRYPOINTNAME is enabled, if it is 0 it is disabled. The entry point is not executed by the scheduler nor from MDL scripts.

kill SESSION,SIGNAL

kill SIGNAL

kill

Kill the session *SESSION* with signal *SIGNAL*. By default the current session is killed with SIGTERM.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert H. de Vries, Dutch Space BV

SEE ALSO

data-monitor(3), mdlAndActions(3), rt-control(3), sched-control(3), EuroSim::Conn(3), EuroSim::MDL(3).

2.2.24 EuroSim::Conn

EuroSim Connection functions

SYNOPSIS

use EuroSim::Conn;

DESCRIPTION

EuroSim::Conn Functions

event_num EVENT_NAME

Return the internal event number given the name *EVENT_NAME* of the event.

event_name EVENT_NUMBER

Return the official event name given the number *EVENT_NUMBER* of the event.

event_list

Return an array of hash tables containing the table of events (messages). Each event has the following hash keys and values: id - numerical event number name - symbolic event name args - string of characters indicating the types of the arguments

sim_connect HOSTNAME, CLIENTNAME, PREFCON, EVENT_HANDLER, SESSION_REF

Internal function used to set up the connection with a running simulator. The *HOSTNAME* and *PREFCON* parameters specify the connection to the simulator. The *CLIENTNAME* is a name used to identify the client in he simulator. The *EVENT_HANDLER* is a reference to a perl function used to process incoming messages. Parameter *SESSION_REF* is passed to the callback as first argument. The function returns a EuroSim::Conn object.

sim_disconnect CONN

Disconnect from a running simulator. The connection is specfied by parameter *CONN*.

session_fd CONN

Return the file handle of the socket of the connection *CONN*.

session_prefcon CONN

Return the connection number of the connection *CONN*.

session_poll CONN

Poll for incoming data on connection *CONN*. For each incoming packet the callback function is called.

rtGo CONN, TV_SEC, TV_NSEC

rtGo CONN

Internal function to generate rtGo event.

rtStop CONN, TV_SEC, TV_NSEC

rtStop CONN

Internal function to generate rtStop event.

rtFreeze CONN, TV_SEC, TV_NSEC

rtFreeze CONN

Internal function to generate rtFreeze event.

rtFreezeAtSimtime CONN, TV_SEC, TV_NSEC

Internal function to generate rtFreezeAtSimtime event.

rtStep CONN

Internal function to generate rtStep event.

rtStepDuration CONN

Internal function to generate rtStep event with a duration.

rtAbort CONN

Internal function to generate rtAbort event.

rtHealth CONN

Internal function to generate rtHealth event.

rtReset CONN

Internal function to generate rtReset event.

maNewMission CONN, SCENARIO_NAME

Internal function to generate maNewMission event.

maOpenMission CONN, SCENARIO_FILE

Internal function to generate maOpenMission event.

maCloseMission CONN, SCENARIO_NAME

Internal function to generate maCloseMission event.

maNewAction CONN, SCENARIO_NAME, ACTION_CODE

Internal function to generate maNewAction event.

maDeleteAction CONN, SCENARIO_NAME, ACTION_NAME

Internal function to generate maDeleteAction event.

maActionExecute CONN, SCENARIO_NAME, ACTION_NAME

Internal function to generate maActionExecute event.

maActionActivate CONN, SCENARIO_NAME, ACTION_NAME

Internal function to generate maActionActivate event.

maActionDeActivate CONN, SCENARIO_NAME, ACTION_NAME

Internal function to generate maActionDeActivate event.

maExecuteCommand CONN, NAME, COMMAND, ACTION_MGR_NR

Internal function to generate maExecuteCommand event.

maSnapshot CONN, FILENAME, COMMENT

Internal function to generate maSnapshot event.

maMark CONN, COMMENT, NUMBER

Internal function to generate maMark event.

maMessage CONN, MESSAGE

Internal function to generate maMessage event.

maRecording CONN, ON_OFF

Internal function to generate maRecording event.

maRecordingSwitch CONN

Internal function to generate maRecordingSwitch event.

maCurrentInitconds CONN, INIT_CONDS

Internal function to generate maCurrentInitconds event.

maCurrentCalibrations CONN, CALIBRATIONS

Internal function to generate maCurrentCalibrations event.

maReload CONN, SNAPFILE, ON_OFF

Internal function to generate maReload event.

dtAdd2LogList CONN, VARNAME

Internal function to generate dtAdd2LogList event.

dtRemoveFromLogList CONN, VARNAME

Internal function to generate dtRemoveFromLogList event.

dtSetValueRequest CONN, VARNAME, VALUE

Internal function to generate dtSetValueRequest event.

dtSetValues CONN, VAR_VALUE_HASH

Internal function to generate dtSetValueRequest event using a hash containing the variable/value pairs.

dtCpuLoadSetPeak CONN, PROCESSOR, PEAK_TIME

Internal function to generate dtCpuLoadSetPeak event.

scSetBrk CONN, TASK_NAME, ENTRY_NR, ENABLE

Internal function to generate scSetBrk event.

scStepTsk CONN

Internal function to generate scStepTsk event.

scContinue CONN

Internal function to generate scContinue event.

scGoRT CONN, ENABLE

Internal function to generate scGoRT event.

scListTasks CONN

Internal function to generate scListTasks event.

scTaskDisable CONN, TASK_NAME, DISABLE

Internal function to generate scTaskDisable event.

scSetTrc CONN, TASK_NAME, ENTRY_NR, ENABLE

Internal function to generate scSetTrc event.

scClearBrks CONN

Internal function to generate scClearBrks event.

scClearTrcs CONN

Internal function to generate scClearTrcs event.

scWhere CONN

Internal function to generate scWhere event.

scListEvents CONN

Internal function to generate scListEvents event.

scRaiseEvent CONN, EVENT_NAME

Internal function to generate scRaiseEvent event.

scSimtime CONN, SIMTIME_SEC, SIMTIME_NSEC

Internal function to generate scSimtime event.

scRaiseEventAt CONN, EVENT_NAME, WALLCLOCK_SEC, WALLCLOCK_NSEC

Internal function to generate scRaiseEventAt event.

scRaiseEventAtSimtime CONN, EVENT_NAME, SIMTIME_SEC, SIMTIME_NSEC

Internal function to generate scRaiseEventAtSimtime event.

scSpeed CONN, ACCELERATION

Internal function to generate scSpeed event.

event_join_channel CONN, CHANNEL

Internal function to join channel *CHANNEL*.

event_leave_channel CONN, CHANNEL

Internal function to leave channel *CHANNEL*.

print_event_list

Print a list of all events (messages) and parameters used in the communication between the test controller and the simulator.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert H. de Vries, Dutch Space BV

2.2.25 EuroSim::Dict

EuroSim Data Dictionary functions

SYNOPSIS

use EuroSim::Dict;

DESCRIPTION

EuroSim::Dict Functions

open FILENAME

Open data dictionary file *FILENAME*. The function returns EuroSim::Dict object.

close DICT

Close the DICT object. Frees memory.

list DICT, PATH

Return a list of child node names of parent node *PATH*.

var_value_get DICT, PATH

Return the numerical value of variable with dictionary path *PATH*.

var_string_get DICT, PATH

Return the string value of variable with dictionary path *PATH*.

var_value_set DICT, PATH, VALUE

Set the numerical value of variable with dictionary path *PATH*.

var_string_set DICT, PATH, VALUE

Set the string value of variable with dictionary path *PATH*.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert H. de Vries, Dutch Space BV

2.2.26 EuroSim::InitCond

EuroSim Initial Condition File functions

SYNOPSIS

use EuroSim::InitCond;

DESCRIPTION

EuroSim::InitCond Functions

read FILENAME, DICT

Read initial condition file with file name *FILENAME*. The values are loaded into an existing *DICT*. The function returns an object of type EuroSim::InitCond.

add INITCOND, FILENAME

Add additional initial condition values from initial condition file with file name *FILENAME* to initial conditions *INITCOND*.

simtime INITCOND

Return the simulation time specified in the initial conditions *INITCOND*.

comment INITCOND

Return the comment string specified in the initial conditions *INITCOND*.

get_varlist_failed INITCOND

Return the list of variables defined in the initial condition file read but not present in the data dictionary. This is used to report to the user that the initial condition file contains variables which are not part of the data dictionary. This is an error.

get_varlist_set INITCOND

Return the list of variables in the initial condition file which were successfully loaded into the data dictionary.

write INITCOND, FILENAME, BINARY

Write out the initial conditions in *INITCOND* to a file named *FILENAME*. If *BINARY* is set to 1, the format is binary, otherwise it is in ASCII format.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert H. de Vries, Dutch Space BV

SEE ALSO

EuroSim::Dict(3)

2.2.27 EuroSim::Link

EuroSim TmTc Link functions

SYNOPSIS

use EuroSim::Link;

DESCRIPTION

EuroSim::Link Functions

open ID, MODE

Open one end of a TmTc Link named *ID* with mode *MODE*. Mode is ”r”, ”w” or ”rw”. The function returns a EuroSim::Link object.

connect LINK, SESSION

Connects one end of link *LINK* to the other end in a running simulator *SESSION*.

close LINK

Close down the link *LINK*.

write LINK, DATA

Write *DATA* to the *LINK*.

read LINK

Read data from the *LINK*. The data read is returned.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert H. de Vries, Dutch Space BV

SEE ALSO

EuroSim::Session(3)

2.2.28 EuroSim::MDL

EuroSim MDL functions

SYNOPSIS

use EuroSim::MDL;

DESCRIPTION

EuroSim::MDL Functions

open FILENAME, DICT

Open the MDL file with file name *FILENAME* and data dictionary *DICT*. The function returns a handle to an MDL object.

close MDL

Close the MDL object.

filename MDL

Return the filename of the MDL file.

action_list MDL

Return a list with the names of all the actions.

script_action NAME, SCRIPT, CONDITION

Create an MDL script text for an action named *NAME* with script action *SCRIPT* and condition *CONDITION*.

monitor_action NAME, VAR1, VAR2, ..., VARN

Create an MDL script text for a monitor action name *NAME* with variables *VAR1*, *VAR2* to *VARN*.

recorder_action NAME, FREQ, VAR1, VAR2, ..., VARN

Create an MDL script text for a monitor action name *NAME* with variables *VAR1*, *VAR2* to *VARN*.

stimulus_action NAME, OPTION, FILENAME, FREQ, VAR1, VAR2, ..., VARN

Create an MDL script text for a stimulus action name *NAME* with variables *VAR1*, *VAR2* to *VARN*.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert H. de Vries, Dutch Space BV

2.2.29 EuroSim::SimDef

EuroSim simulation definition functions

SYNOPSIS

use EuroSim::SimDef;

DESCRIPTION

EuroSim::SimDef Functions

open FILENAME [, HOSTNAME]

Open the simulation definition file with file name *FILENAME*. By default the local host is used to determine the simulator and dict paths. By specifying a specific *HOSTNAME* you can override this. The function returns a handle to a simulation definition object.

close SIM

Close the simulation definition object.

filename SIM

Return the filename of the simulation definition file.

mdl_list SIM

Return an array with filenames of MDL files.

model SIM

Return the filename of the model file.

initcond_list SIM

Return an array with filenames of initial condition files.

calibration_list SIM

Return an array with filenames of calibration files.

schedule SIM

Return the filename of the schedule file.

dict SIM

Return the filename of the data dictionary file.

outputdir SIM, OUTPUTDIR

Set the output directory.

workdir SIM, WORKDIR

Set the work directory.

alias SIM, ALIAS

Set the alias file.

tsp_map SIM, TSP_MAP

Set the TSP map file.

update_arch SIM, HOSTNAME

Set the dict and simulator paths correctly based on the hostname.

refresh_outputdir SIM

Refresh the output directory name based on the current date and time.

realtime SIM [,RT_FLAG]

Get/Set the realtime flag. 0 = non-realtime, 1 = real-time.

launch HOSTNAME

Launch the simulator.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert H. de Vries, Dutch Space BV

2.2.30 EuroSim::Window

EuroSim Test Controller window

SYNOPSIS

use EuroSim::Window;

DESCRIPTION

EuroSim::Window Functions

new SESSION

Create a new test controller window. Used in interactive mode to display messages coming from the simulator. It uses a FIFO to communicate send incoming messages to the window.

DESTROY

Cleanup the FIFO file.

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert H. de Vries, Dutch Space BV

2.3 man4: special files & file formats

This section of manual pages contains descriptions of selected file formats read/written by the tooling in the EuroSim distribution.

2.3.1 api

Application Programming Interface syntax

DESCRIPTION

The API information is stored in the submodel or compilation unit in commented headers. Coding languages that only support **one-liner comments** (eg **Fortran 77**, **ADA**) should obey the following rules :

1.

The **API** information should be one continues block of comments, **lines with only comment delimeters** are allowed.

2.

The **API** information should be ended with an empty line.

API format is described either in YACC(1)-type format, -or- extended Backus-Naur format, -or- my own interpretation of both.
Keywords are in **boldface**. User specified text is in *italics*. All language compound constructs start with a Capital. Optional arguments are put between [ ]. Constructs that might occur more than once, are directly followed by '..'.
EuroSim Application Programming Interface syntax:
'Global_State_Variables
type name[**:** Variable_attribute ..]
[**,**

type name[**:** Variable_attribute ..]] ..

'Global_Input_Variables
type name[**:** Variable_attribute ..]
[**,**

type name[**:** Variable_attribute ..]] ..

'Global_Output_Variables
type name[**:** Variable_attribute ..]
[**,**

type name[**:** Variable_attribute ..]] ..

'Entry_Point *entry* [()] [**:** Entry_attribute ..]
[**'Global_Input_Variables**

type name[**:** Variable_attribute ..]

[**,**

type name[**:** Variable_attribute ..]] ..

[**'Global_Output_Variables**

type name[**:** Variable_attribute ..]

[**,**
type name[**:** Variable_attribute ..]] ..

Elaboration of compound constructs :

Entry_attribute -> **DESCRIPTION="** *free* text **"**
Entry_attribute -> **TIMING="** *int* **,** *int* **,** *int* **"**

Variable_attribute -> **DESCRIPTION="** *free* text **"**
Variable_attribute -> **UNIT="** *free* text **"**
Variable_attribute -> **INIT="** *value* .. **"**
Variable_attribute -> **MIN="** *value* .. **"**
Variable_attribute -> **MAX="** *value* .. **"**
Variable_attribute -> **PARAMETER** | **RO**

Each attribute can be mentioned only once per variable or entry respectively.

SEE ALSO

api2dict(1), **dict2api**(1), **subm2dict**(1)

BUGS

The meaning/purpose **keywords** and some **user specified text** are not explained.
Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Ad Schijven, Dutch Space BV, The Netherlands.

2.3.2 alias

Alias file format

DESCRIPTION

An alias file is used to create aliases of data dictionary variables. The alias file is loaded at simulator startup time. The aliases are placed under an /alias node in the data dictionary tree.
Empty lines are ignored. Everything after a # character is ignored.
Every non-skipped line contains the following information:

```
Alias       Alias name
Path        Data dictionary path of a variable.
```

The items within a line are separated by white space; each line is separated from the next by a new-line. The alias name must begin with an alphabetic character and must be followed by 0 or more alphanumerical characters or underscores.

EXAMPLE

```
#
# Example alias file
#
XCAD1002   /SPARC/ControlStatus
XZCY2945   /SPARC/setpoints

a          /SPARC/setpoints[2]
b_2        /SPARC/setpoints[3][3]
```

SEE ALSO

dictAliasOpen(3).

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Robert de Vries (DS)

2.3.3 dictdump, initcond, snapshot

EuroSim initial condition and snapshot file format

DESCRIPTION

Initial condition files are ASCII or binary files, read and written with ***dictdump*(3)** functions. The extension is either *.snap* or *.init*.
The file consists of a *header* section and a *data* section. Empty lines and lines starting with a "#" character in the *header* section are in principle skipped as being 'comment lines', however when the rest of the line following a "#" character contains a valid keyword/value pair, it is interpreted. These lines have the form:
 # keyword = value
Used and thus valid keywords are:

comment

the comment that was passed when the file was written. May be ommitted.

format

either *ascii* or *binary*. When ommited the file is interpreted as being *ascii*.

simtime

the simulation time at the time the snapshot was taken.

dict

the EuroSim data dictionary file from which this file was written. The path to the datadictionary is relative to the place where it can be found. May be ommited.

reference

an optional version control reference for the state of the model this file's datadictionary was generated from.

Any other keywords can be generated by ***dictdump*(3)**, or by the user, but they are not interpreted.
Every initial condition or snapshot file *written* by EuroSim also contains a comment line indicating the type of snapshot or initial condition file written. It is either:

```
# contains only differences wrt dict default values
```

-or-

```
# contains all current dict values at Mon Jan 27 14:15:24 1997
```

which indicates whether it is a partial snapshot (or initial condition) file or a complete snapshot containing all the variables in the data dictionary.
When the **format** of the file is *binary* there is at least one mandatory empty line following the header.
The *data* section of a *binary* file contains records for each datadictionary symbol as follows:
{ **symbol_length+1, symbol, value_length, value** }
where the *symbol*s are fully qualified data dictionary paths and the *value*s for the symbols are of course in 'binary' form (no formatting).
When the **format** of the file is *ascii* the records of the *data* section look like:
{ **"InitialCondition: ", symbol, "=", value** }
again the *symbol*s are fully qualified data dictionary paths, the *value*s for the symbols are formatted. The records may extend several lines but the carriage return '\n' is then escaped with a '\' backslash, so in there is in principle one record per line.

EXAMPLE

The following example shows a typical layout of a *full* (ASCII) initial condition file:

```
# EuroSim initial condition file
# version  = @(#)Header: dumpfile
# dict     = thermo.dict
# comment  = complete ascii dump
# format   = ascii
# contains all current dict values at Mon Jan 27 14:15:24 1997
#
InitialCondition: /thermo.f/thermo$celltemp = "{ { 0, 0, 0},\
{ 0, 0, 0}, { 0, 0, 0}, { 0, 0, 0}}"
InitialCondition: /thermo.f/initthermo/thermo$capa = "{ { 0,\
0, 0}, { 0, 0, 0}, { 0, 0, 0}, { 0, 0, 0}}"
InitialCondition: /thermo.f/initthermo/thermo$condfac = "0"
InitialCondition: /thermo.f/initthermo/thermo$emisfac = "0"
```

SEE ALSO

dictdump(3).

BUGS

dictdump(3) does not do free format files (without the ”InitialCondition: ” keyword?) yet.
Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Wim Elhorst (Dutch Space BV)

2.3.4 model

Model file format

DESCRIPTION

Model files are ASCII files, created and interpreted with the help **dictmodel**(3) operations. Empty lines and lines starting with a "#" character are skipped.
Every non-skipped line contains the following information:

```
fileID          - may be empty
filetype-descr  - may be empty
model-path
description     - may be empty
dependency      - may be empty
```

The items within a line are separated by colons; each line is separated from the next by a new-line; but the new-line may be escaped by a back-slash, thus creating larger records....

EXAMPLE

```
# This is the "model definition" for thermo.model
#

:FreeText:/__MODEL_type:binary
:FreeText:/__MODEL_target:thermo.exe
:FreeText:/__MODEL_idir:
:FreeText:/__MODEL_libs:
Model_description 1.2:TXT_FILE:/Model_Description:
thermo.schedule 1.2:SCHEDULE_FILE:/Schedule:
obs.c :C_FILE:/SPARC:
:ORG:/Hardware:
heaters.c 1.2:C_FILE:/Hardware/heaters.c:
sensors.c 2.2:C_FILE:/Hardware/sensors.c:
sensors.c:DEP::sensors.h
thermo.f 1.1:F77_FILE:/Hardware/thermo.f:
model.env 1.2:ENV_FILE:/model.env:
```

SEE ALSO

dictmodel(3).

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Jaap Schuttevaer (ACE)

2.3.5 dictrec

EuroSim time series/recording/stimulus file format

DESCRIPTION

EuroSim recording files are binary files with an ASCII header, read and written with ***dictrec*(3)** functions. The file name extension is *.rec*.
The file consists of a *header* section and a *data* section. Empty lines and lines starting with a "#" character in the *header* section are skipped as being 'comment lines'.
The *header* section consists of key/value pairs and 'variable name'/'type shape' pairs separated by colons (:). The ordering of the key/value pairs is irrelevant. The 'variable name'/'type shape' sublist is ordered and should always follow the **Number of variables** key.
Valid keywords are:

Dict

the EuroSim data dictionary file from which this file was written. The path to the datadictionary is relative to the place where it can be found. May be ommited.

SimTime

the name used for the simulation time in the 'variable name'/'type shape' sublist. Iff this field is ommited from the *header* section it is assumed that the simulation time was not part of this stimulus file.

TimeFormat

'UTC' or 'relative' time.

Record size

the length (in bytes) of the time serie records on this file.

Version

the file format version of this file.

Mission

an optional version control reference for the mission this file was generated with.

Date recorded

an optional indication for the date this file was recorded with EuroSim.

Reference

an optional version control reference for the state of the model this file's datadictionary was generated from.

Number of variables

gives the number of variables recorded in this file. The key/value should always be followed by the 'variable name'/'type shape' sublist. Note that the variables recorded are in principle only scalar variables and array variables of base types.

The 'variable name'/'type shape' sublist contains records as follows:
<*variable-path-name*>**:**<*basetype-name*>[*optional-array-dimensions*] [, **offset=***optional-byteoffset*]
with the *basetype-name*'s being *C*-basetypes, (eg. *double*, *int*, *unsigned short*, etc...).
The *optional-array-dimensions* are specified just like in the *C* language with square brackets per dimension.
The **offset=** keyword followed by the *optional-byteoffset* corrects for missaligned data. Missaligned data can occur in recorded *C*-structs. Offsets are only written if an offset correction is deemed necessary, eg.

when the size of a former variable's *basetype-name* does not equate with the next expected byte offset for the raw binary data.
The binary *data* section of the file contains records with a bytesize of **Record size**, consisting of a binary dump of values of the objects listed in the 'variable name'/'type shape' sublist.

EXAMPLE

The following example shows a typical layout of a binary recording file:

```
# EuroSim recording file
Version: Mk0.2-rev2
Date recorded: Mon-Jul-14-15:37:07-1997
Mission: yet-another.mdl
Reference: @(#)$Trace: yet-another.model (not_traceable) $
Record size: 71
Dict: yet-another.dict
SimTime: /simulation_time
TimeFormat: relative
Number of variables: 8
/simulation_time: double
/varsForRec/initVars/testArray: double[2][3]
/varsForRec/initVars/testSimple: unsigned short
/varsForRec/initVars/testStruct.a: int
/varsForRec/initVars/testStruct.b: double
/varsForRec/initVars/testStruct.c: char
```

This header is followed by raw binary data.

SEE ALSO

dictrec(3), **r2a**(1), **a2r**(1).

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Wim Elhorst (Dutch Space BV)

2.3.6 exports

EuroSim External Simulator Access exports file

SYNOPSIS

$PROJECT_DIR/$MODEL.exports

DESCRIPTION

The filename of the **exports** file is derived from the model filename. It is constructed by replacing the .model extension with a .exports extension. It is placed in the same directory as the model file.
This file describes which parts of the EuroSim data dictionary may be accessed by external (non-EuroSim) simulators. For each part that should be accessible for external simulators, one can indicate how it can be accessed (read / write or both) and by whom.
The **exports** file consists out of a number of lines, each line describing one part of the data dictionary that may be exported. Empty lines, and lines beginning with a '#' are ignored. Data following a '#' is assumed to be comment and is ignored. Each non-empty line has the following layout:

path

The path to the data dictionary which should be exported

id

The name under which this path should be exported

mode

Operation that can be performed on this part of the data dict (read, write or both)

users

The clients that may request access

The given path that is exported means that every subtree or variable that is located underneath that path may be requested in a view. A simple way therefore to export every variable in the data dictionary is to export the '/'. Since this is the root of each data dictionary, every view is now allowed.
The id under which the part of the data dictionary is exported is the name which the external simulator must use in his access-request (see **extViewCreate**() in ***extView*(3)**). This id is introduced to achieve some information-hiding. The external simulator needs to know nothing about the position of the view in the data dictionary.
The mode of the export can have three values.

R

for reading only.

W

for writing only.

RW

for both reading and writing.(Not Yet Implemented)

The choice for Read Write access has some consequences and before doing this, one should read ***extRW*(8)**.
The user's argument is used to allow only certain simulators access to the data dictionary or create trusted hosts. When there are no users listed, this rule will apply for all users.

EXAMPLE OF AN EXPORTS-FILE

The following output is a example of an exports file for a EuroSim simulator.

```
#
# An example of an exports file
#
/space/station/era      era         R
/space/stars            stars       RW
/space/rockets/ariane   esarocket   W
```

SEE ALSO

extView(3), **extClient**(3), **extExport**(5), **extRW**(8).

BUGS

Using extra '/' characters in the base-path confuse the current dict-functions. It is therefore wise not to add extra '/' characters in the paths of the **exports** file.
The users that apply for a rule are for now ignored. Each rule applies for all users.
The **RW** option in the exports file is not yet implemented because of the synchronization problems. Adding a data dictionary path for **'R'** and **'W'** mode on separate lines is however possible. The names of the read-view and write-view should be different.
Mail bug reports to esim-bugs@dutchspace.nl.

Author

Robert de Vries, Dutch Space BV, The Netherlands.

2.3.7 esimcapability

EuroSim capability database format

DESCRIPTION

The EuroSim capability database format is a very simple ASCII database read by **esimcapability**(1). It is used to generate include file and library locations and generate code from the graphical EuroSim tooling. In the file a capability (key) can be defined together with its associated data. Each non-empty line should contain a statement in the form:
key = value
Used and thus valid keywords are:

capability

the name of the capability, must be the first line of a capability description

Per **capability** key the following keys can be specified or ommited. The default value for an ommited key is just the empty string:

os

the operating system(s) for which this capability is available. This may also be a space separated list, as in: *IRIX-5.3 IRIX-6.x*.

includepaths

the includepaths which are neccesary for this capability

librarypaths

the librarypaths which are neccesary for this capability

libs

the libraries which are neccesary for this capability

defines

the defines which will be passed on to the compiler

descr

description of this capability. Each capability must include this field once

value

optional 'string' value for this capability.

The case of the keys is irrelevant, as the **auxcapability**(3) functions ignore it. Lines starting with a '#' or empty lines are considered comments and are ignored.

EXAMPLE

The following example shows the 'default capability'; eg. the definition and locations of the EuroSim runtime libraries.

```
capability   = Mk02
OS           = IRIX-5.3
includepaths = $EFOROOT/include $EFOROOT/include/esim
librarypaths = $EFOROOT/lib
libs         = esServer
defines      = __eurosim__
descr        = EuroSim run time library
```


SEE ALSO

esimcapability(1), **auxcapability**(3).

BUGS

There should be better control over os dependent features and flags.
Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Wim Elhorst (Dutch Space BV)

2.3.8 plt

Plot library file format specification.

DESCRIPTION

During a test analyse session a list of graphs can be generated, which must be plotted by the backend. The test analyse session can be repeated in the future if the list of graphs *graphList* is stored to file with **pltWrite**. The stored session can be read from file with **pltRead**.
The graphlist is stored to a file with a specific file format which will be explained in the following. Each attribute of a graph is stored with the following syntax: *keyword* <comma seperated value list>;
For example grid_style 5; canvas_position 10,10;
Comment lines start with "#".
Attributes are grouped. The syntax of a group is given by *keyword* [identifier] { *attribute list or group* }
The identifier is an optional string e.g.: graph "graph_1" { file_description "file 1"; canvas_position 10,10; }
The order of the group and attributes within groups are not fixed.
In the following each group with its attributes will be explained.

GROUP header

The group **header** has the following attributes or groups:

version

file version.

GROUP graph "graph_id"

The group **graph** defines a graph with title graph_id, which is a string. The group **graph** has the following attributes:

file_description "string";

Description of file.

plot_description "string";

Description of plot.

canvas_position xpos,ypos;

Canvas position of graph icon. The arguments xpos and ypos are integers and specify the X and Y position on the canvas.

legend_position top_left|top_right|bottom_left|top_right;

This attribute sets the legend position.

group grid

group axis

group time

group variable

group curve

GROUP grid

This group is only allowed within group **graph** and has the following attributes:

enabled true | **false;**

Enable or disable the drawing of a grid in the plot.

grid_style ***integer*****;**

Specify the grid style. Specifying the grid style makes only sense if the grid is enabled.

GROUP time

This group is only allowed within group **graph**. With this group one can select a time window. All data recorded during this time window will be plotted. The following attributes are defined:

auto_mode_min true | **false;**

Determine the minimum value of the time window automatically (**true**) or manually (**false**). If manually determined then one must also specify **min**.

min ***double*****;**

Specify the lower range of the time window.

auto_mode_max true | **false;**

Determine the maximum value of the time window automatically (**true**) or manually (**false**). If manually determined then one must also specify **max**.

max ***double*****;**

Specify the upper range of the time window.

GROUP axis "axis_id"

This group is only allowed within group **graph**. With this group one can define the axis specified by the axis id. The axis id is a string. Valid string values are "x" for the X-axis, "y" for the Y-axis, "x2" for the X2-axis, "y2" for the Y2-axis. The following attributes are defined:

scaling linear | **logarithmic;**

Set the axis scaling to linear or logarithmic.

label "string" | <**automatic**>**;**

Set the axis label to the specified string or let the system generate a label automatically.

group range

GROUP range

This group is only allowed within group **axis**. With this group one can define the axis range. The following attributes are defined:

auto_mode_min true | **false;**

Determine the minimum value of the axis range automatically (**true**) or manually (**false**). If manually determined then one must also specify **min**.

min ***double*****;**

Specify the lower axis range.

auto_mode_max true | false;

Determine the maximum value of the axis range automatically (**true**) or manually (**false**). If manually determined then one must also specify **max**.

max *double*;

Specify the upper axis range.

GROUP variable *variable_id*

This group is only allowed within group **graph**. With this group one can define variables used for the plotting. The variable_id identifier is a string. A variable id string must start with '$' followed by a number. The number of the variables within a **graph** group must be consecutive. The following attributes are defined:

recorder_file_abs "string";

Set the filename of the recorder file, where the variable is stored. The filename must be a absolute path.

recorder_file_rel "string";

Set the filename of the recorder file, where the variable is stored. The filename is a relative path. If **recorder_file_rel** is specified, then it does not make sense to specify **recorder_file_abs**, because this attribute will be determined run time.

column *integer*;

Specifies the column number of the variable in the recorder file.

var_path "string";

Specifies the variable name in the recorder file. Note that it does not make sens to specify both the **column** attribute and the **var_path** attribute, because the column number will be overwritten if the **var_path** attribute is specified.

GROUP curve

This group is only allowed within group **graph**. With this group one can define the curves in a plot. The following attributes are defined:

pattern *integer*;

Specifies the line pattern.

legend "string" | <**var_name**> | <**var_path**> | <**var_descr**>**;**

Specifies the legend belonging to the curve. The legend can be one of the predefined built-in types: <var_name>, <var_path> and <var_descr> or a user-defined string "string". <var_name> gives you only the name of the variable (the default). <var_path> gives you the complete EuroSim data dictionary path name. <var_descr> gives you the variable description from the data dictionary.

axis "axis_id" "function_definition";

This attribute defines the coordinates of the curve. Valid axis id's are "x", "y", "x2", "y2". The function definition is a string and can be a variable or a function of variables.

EXAMPLE

This is an example of a plot definition file:

```
header   {
   version  1.0;
}
graph  "graph_1"   {
   file_description "file 1";
   canvas_position  10,10;
   plot_description "plot 1";
   legend_position  top_right;
   grid {
      enabled true;
      grid_style 5;
   }
   time {
      auto_mode_min true;
      auto_mode_max true;
   }
   axis "x" {
      scaling linear;
      label   "x axis graph 1";
      range {
         auto_mode_min true;
         auto_mode_max true;
      }
   }
   axis "x2" {
      scaling logarithmic;
      label   "x2 axis graph 1";
      range {
         auto_mode_min false;
         min       -10e-5;
         auto_mode_max false;
         max        10e5;
      }
   }
   axis "y" {
      scaling logarithmic;
      label   <automatic>;
      range {
         auto_mode_min false;
         min       -10e-5;
      }
   }
   axis "y2" {
      scaling linear;
      label   "y2 axis graph 1";
      range {
         auto_mode_max false;
         max       -10e-5;
      }
   }
```

```
    variable "$1" {
       recorder_file_abs  "/tmp/junk";
       var_path           "//dir1/dir2/abs";
    }
    variable "$2" {
       recorder_file_abs  "/tmp/junk";
       column             3;
    }
    variable "$3" {
       recorder_file_rel  "/tmp/rel";
       var_path           "//dir1/dir2/rel";
    }
    curve {
       pattern    5;
       legend     <var_descr>;
       axis  "x"  "$1";
       axis  "y"  "$2";
    }
    curve {
       legend     "curve_2";
       axis  "x2"  "$1";
       axis  "y2"  "$1*$1";
    }
    curve {
       legend   "curve_3";
       axis  "x2"  "log($1)";
       axis  "y2"  "exp($1*$1)";
    }
}
```

SEE ALSO

pltWrite(3), **pltRead**(3).

AUTHOR

Guo Tschi, Dutch Space BV, The Netherlands.

2.4 man8: daemons, doumentation & maintenance commands

2.4.1 extRW

Explains the restrictions of Read Write access.

DESCRIPTION

When exporting parts of the database, it is recommended to allow only read **or** write access. The use of read and write access will impose problems that can (for now) not be solved. When simulator A and B share their data dictionaries with both read and write access, and they both change variable 'var' at the same moment the data dictionaries will become out of sync. For example:
Simulator A changes variable 'var' to value 1, and simulator B changes the value to 0. They both send that change to the other simulator, and when that data reaches the other simulator, simulator A will set his variable 'var' to 0 (since Simulator B requested that) and simulator B will set his var to 1.
The result is that the data dictionaries are out of sync, and will remain so until one of the simulators changes the value again. The chance that this will happen is very big since the critical area starts when the data is changed, and it ends when the data is set at the other side. Depending on the frequency of the updates and the performance of the network, this might take several hundred milliseconds.

RECOMMENDED APPROACH

The recommended approach is to split the data into two parts. Each simulator should have a data-area that he alone will update, and another part in which he can see the status of the other simulator. According to that status each simulator can decide himself what he will do with that information. This way data will never be overwritten.

FUTURE SOLUTIONS

This proposed solution might work for two coupled simulators (although this is an overhead in simulation-code) but the problems expand with each extra simulator that wants to share the same data. It is possible to fix the problem described above, but it will take a few days to implement, and it will take some computing-power and extra memory at the both the external simulator and the EuroSim simulator.
Since each update is accompanied with a timestamp, it is possible to remember for each variable its last update-time. Although changed data is only discovered when the update is being sent (and not when the data is actually updated (by directly accessing the data-area)) this time-stamp will be a little bit rough, but it will do. When an update then comes in, we can check for each variable when the last update occurred and whether this data is more recent. The chance that the time-stamps are equal is practically zero since the time-stamps are noted down in micro-seconds.
A last adjustment that can be done to improve the accuracy of the timestamps is by allowing the 'user' (simulator) himself to indicate that the value is recently changed. This should not be done for each variable since it gives a reasonable overhead, but it may be necessary for some critical data to know the exact time that is changed.

SEE ALSO

extExport(3).

BUGS

Mail bug reports to esim-bugs@dutchspace.nl.

AUTHOR

Erik de Rijk, Dutch Space BV, The Netherlands.

2.4.2 esimd

Daemon for EuroSim simulation processes

SYNOPSIS

esimd [**-v**] [**-i**] [**-n**] [**-c**] [**-p**] [**-l** *logfile*] [**-L** *licensefile*] [**-f** *freq*] [**-d** *period*]

DESCRIPTION

esimd is a daemon for interfacing with EuroSim simulation processes and configuring the target hardware.
A machine in a network running this daemon is called a *EuroSim Simulation Server*: it can run simulators which have linked in the EuroSim runtime library (See **esim**(3)).
The functions of **esimd** are fivefold:

- It checks whether the host is licensed to run the daemon. A license file can be specified through the *-L* option. If no file is specified per default the file *$EFOROOT/etc/EuroSim.licenses* is checked. The daemon will not start if there is no valid license.
- It provides server status information over the network for potential EuroSim clients. This includes information on whether a hosts can be used as a *Simulation Server*, and what simulators (i.e. executing simulations) are already running on that host.
- On request, it launches a simulator for a particular EuroSim client, after checking the validity of the request. It assembles and verifies the complete request of any client that requests the launch of a simulator.
- It configures (when on a multiprocessor machine) the processors for real-time usage. When an existing simulator has already been launched in real-time executing mode on that host, a second real-time simulator is not possible. When the simulator that claimed the real-time executing mode finishes the machine is reconfigured back to normal (See **mpadmin**(1) for details on isolated, restricted & non-preemptive processors).
- It routes all diagnostic information from the simulator to either ”/dev/console” or a logging file (*-l* option).

Esimd reopens its log file when it receives a hangup signal, SIGHUP. The log file may be moved before the log file is reopened. This can be used in crontab jobs to cycle log files.
The client side of the protocol for querying EuroSim Simulation Servers and launching simulators is covered in ”C” (See **evc**(3)).
The daemon is based on an RPC based protocol for Mk0.2 but is also backward compatible with Mk0.1 ScenarioManagers and simulators.

OPTIONS

-v

Verbose. *esimd* operates in verbose mode, mostly telling you what it’s doing, and logging these messages through the syslog daemon (see **syslogd**(1M)). It sends the messages through the LOG_LOCAL0 channel (see **syslog**(3)). When *-v* is specified, this flag will also be passed to the simulator, causing it to send diagnostic information either to ”/dev/console” or to the logfile specified with *-l*.

-i

initial condition indicates that an initial condition file is to be applied when the requested simulator is launched *before* it reaches ”initialising” state.

-n

This flag prevents the daemon from backgrounding itself. Useful when debugging with purify and dbx.

-p

This flag prints the process id of the daemon started.

-c

This flag provides backward compatibility for older Mk0.2 and Mk0.1 simulators when allocating real-time processors. The backward compatible behaviour is that when a simulator was started with the real-time flag on, automatically all processors minus one are restricted, isolated and set non-preemptive. When this flag is not given, processors are only switched to this state on request of the scheduler, when the simulator is in real-time mode.

-l *logfile*

specifies the logfile to which diagnostic information from the launched simulators will be written. Be sure to include logging information from this file in your bug reports. If no logfile is specified all diagnostic information is written to */dev/console*.

-L *licensefile*

specifies the license file where the EuroSim license information can be found.

-f *freq*

asynchronous frequency to run with. Passes the specified asynchronous frequency to the launched simulator. See **esim**(3) or **actionmgr**(3) for details.

-d *period*

time period for datalogger (wrt to specified asynchronous frequency) for datalogger in actionmanager. See **esim**(3) or **actionmgr**(3) for details.

NOTES

Super-user (root) privileges are needed to start up the daemon. Typical usage is:

```
# esimd -v -i -l /var/log/esimd.log
```

A shell script is contained in the EuroSim distribution to put into ”/etc/init.d” for auto-startup of this daemon. Look for the file ”esim.init.d” in $EFOROOT/etc and modify and copy it into ”/etc/init.d”.

BUGS

The daemon currently listens on the UDP and TCP ports ”esim-brdcst” as listed in ”/etc/services” (or NIS), and responds to all queries on these ports. Hence, it might pose a security problem in a vulnerable network environment.
The logfile specified with the *-l* option is written to root (/) and not verified, when no complete file path is specified.
Send bug reports to the EuroSim development team by using the spr tool, or mail them to esim-bugs@dutchspace.nl.

SEE ALSO

mpadmin(1), **efoList**(1), **efoKill**(1), **esim**(3), **services**(4).
And for EuroSim development interfaces: **actionmgr**(3) and **evc**(3).

AUTHOR

Erik de Rijk, Wim Elhorst and R.H. de Vries for Dutch Space BV, The Netherlands.

Appendix A

Abbreviations

Abbreviation	Description
ADD	Architectural Design Document
AD/R	Architectural Design Review
AIFF	Audio InterFace Format
ANSI	American National Standards Institute
API	Application Program Interface
ASCII	American Standard Character Information Interchange
COTS	Commercial Off The Shelf
DDD	Detailed Design Document
DD/R	Detailed Design Review
ESA	European Space Agency
FSS	Fokker Space & Systems
GUI	Graphical User Interface
HIL	Hardware In the Loop
HOOD	Hierarchical Object Oriented Design
ICD	Interface Control Document
SCMP	Software Configuration Management Plan
SGI	Silicon Graphics Inc.
SPMP	Software Project Management Plan
SQAP	Software Quality Assurance Plan
SWSTD	Software Standards
SUM	Software User Manual
TBD	To Be Defined
TBW	To Be Written